ZX Spectrum User's Handbook

R J Simpson & T J Terrell

Newnes Microcomputer Books

ZX Spectrum User's Handbook

Robert J. Simpson and Trevor J. Terrell

Systems and Instrumentation Division
Preston Polytechnic

Newnes Technical Books

Newnes Technical Books
is an imprint of the Butterworth Group
which has principal offices in
London, Boston, Durban, Singapore, Sydney, Toronto, Wellington

First published 1983

British Library Cataloguing in Publication Data

Simpson, R. J.
ZX spectrum user's handbook.
1. Sinclair ZX Spectrum (Computer)—Handbooks, manuals, etc.
I. Title II. Terrell, T. J.
001.64'04 QA76.8.S/

ISBN 0-408-01323-0

Filmset by Tunbridge Wells Typesetting Services Ltd.
Printed in England by Butler & Tanner Ltd., Frome and London

Preface

Computers have developed rapidly since they were described by the British mathematician A. M. Turing in the 1930s and the subsequent demonstration of Eniac (electronic numerical integrator and calculator) by T. P. Eckert and J. W. Mauchly at the University of Pennsylvania in the 1940s. Present day microelectronics technology has enabled the development of compact, inexpensive computers with considerable sophistication, and their effects on industry, commerce, education and everyday life are far-reaching.

The availability of the ZX Spectrum has brought the microcomputer within everyone's reach. However, some of the facilities offered by the Spectrum may be bewildering to many users and, consequently, the purpose of this book is to explain the features, programming aspects and operation of the ZX Spectrum and its peripherals. The book also provides the user with information about the microcomputer so that it may be used efficiently and its features exploited in practical applications.

The book covers troublesome aspects of Basic programming, colour graphics, principles of machine code programming, hardware details and principles of interfacing user hardware via the edge connector. Appropriate worked examples, exercises and demonstration Basic and machine-code programs are included to enable you to gain hands-on experience and to permit you to investigate and verify points explained in the text. The book should also serve as a useful source of reference when programming and using the ZX Spectrum computer.

We wish to thank Sue Wasson for her hard work and competence in typing the manuscript. We are grateful to United Technologies Mostek for their kind permission to use the Z80A Instruction Set summary from their Data Book. We also thank Tom Izatt and our daughters Andrea (S), Janet (T) and Lesley (T) for helping us to

understand the rudiments of the musical notes generated by the ZX Spectrum. We also express our thanks to David Platt for his assistance in building the audio amplifier and the one-byte memory-mapped interface.

We sincerely thank our wives Meryl (S) and Jennifer (T) for their unfailing support and encouragement.

Robert J. Simpson
October 1982 Trevor J. Terrell

To Jennifer and Meryl
and to Andrea, Janet and Lesley

Contents

1

Hands-on

Introduction

You may start working with your ZX Spectrum by connecting the d.c. power supply (jack plug) to the 9V d.c. socket, and by connecting the UHF TV cable from the UHF aerial input of your colour or black and white set to the TV socket on the Spectrum.

After switching on the d.c. supply and TV set, select a TV channel and tune it until the caption

© 1982 Sinclair Research Ltd

appears in black on the lower part of the white TV screen. You will probably find it desirable to turn your TV volume control to its minimum setting when working with your computer.

As a simple test to establish that your computer is working correctly, we suggest you press the following keys in the order given. First press **PRINT**, then 9, then + (this is obtained by holding down the **SYMBOL SHIFT** key and pressing K), then 8, and finally **ENTER**. The computer then calculates the sum 9+8 and displays the result, 17, in the top left-hand corner of the screen. The displayed report code message, in the lower left-hand part of the screen, will be

Ø OK, Ø:1

indicating successful completion of the instruction (a full list of report codes is given in Table 1.1).

Now that you have got started try a few more sums to help you become familiar with the keyboard and the display.

The keyboard (Plate 1.1)

There are up to six characters, symbols and words assigned to each of

the 40 keys, and all the keys except **CAPS SHIFT** and **SYMBOL SHIFT** have repeat capability, that is if the key is held down for more than approximately one second, then it continuously repeats until the key is released. Normally the computer displays a flashing cursor to prompt you to key a valid entry.

After switching on your ZX Spectrum and obtaining a display of the Sinclair caption, press any single key except **CAPS SHIFT** or **SYMBOL SHIFT**. You will notice that the caption disappears and, if a key with a number Ø to 9 or **SPACE** is pressed, then the number or space appears on the screen followed by a flashing K; this is the keyword mode cursor **K**. In contrast, if a statement key is pressed the keyword appears on the screen, followed by a flashing L; this is the letter mode cursor **L**.

The **K** cursor is always displayed when the system requires you to enter either a program line number (any positive integer in the range 1 to 9999) or a keyword (the words written on, above or below the individual keys). When entering programs the **K** cursor occurs automatically at places in the program line where the ZX Spectrum is expecting a valid keyword.

After entering a keyword the computer displays the **L** cursor to indicate that it is waiting for you to input an alphanumeric symbol (number or letter), **SPACE** or **ENTER**, as shown in white on the individual keys, or to input an appropriate symbol or keyword shown in red on the individual keys. The alphabetic characters entered and displayed when in **L** mode are always lower case unless the **CAPS SHIFT** key is also pressed. Upper case alphabetic characters or the digits Ø to 9 may be obtained all the time by entering the capitals mode, which is indicated by the capitals mode cursor, a flashing C. The **C** mode can be obtained by pressing keys **CAPS SHIFT** and 2 to achieve **CAPS LOCK**. The computer will remain in the **C** mode instead of **L** mode until you request a return to **L** mode by again pressing **CAPS SHIFT** and 2. The symbol or keyword shown in red on the keys is obtained by holding down the **SYMBOL SHIFT** key (lower right-hand corner of the keyboard) and pressing the individual key.

The extended mode, indicated by the flashing **E** cursor, is obtained by holding down the **CAPS SHIFT** key and then pressing the **SYMBOL SHIFT** key. In this mode the statement shown above and below the individual keys may be selected, shown on the keyboard in green and red respectively. To use the statements shown in green above the key you must first obtain the extended mode cursor **E** and then press the appropriate key, whereas to use the statements shown in red below the key you must obtain the **E** cursor, then hold down the **SYMBOL SHIFT** key and press the appropriate key.

If you try to enter commands or data which are not valid in ZX Spectrum Basic, the microcomputer interpreter and operating

system, which is monitoring your actions, will indicate that a mistake has occurred by displaying the flashing syntax cursor **?** next to your error. To erase the error you can repeatedly use the **DELETE** key (**CAPS SHIFT** and 0) to remove the part of the line up to and including the **?** cursor, and then enter the correct Basic under the re-established cursor mode. Alternatively you can shift the cursor left, using ◊ (**CAPS SHIFT** and 5), or right, using ◊ (**CAPS SHIFT** and 8), to the point in the program line where the error exists. The mistake can then be deleted using the **DELETE** key and new text keyed in.

To enter graphic symbols or valid inverse characters in your Basic program you must select the Graphics mode. To establish this mode you press the **GRAPHICS** key (**CAPS SHIFT** and 9) and the flashing graphics cursor **G** will be displayed. This enables you to enter any number of the 16 graphics symbols (see codes 128 to 143 in Table 7.1) and any number of the 21 user-defined graphics symbols (see Chapter 7). To obtain a normal graphics symbol (with codes 128 to 135) the appropriate key 1 to 8 is pressed, and to obtain an inverse graphic symbol (with codes 136 to 143) the **CAPS SHIFT** key and the appropriate symbol key (1 to 8) is pressed. To obtain a user-defined graphic symbol, which must first be defined as explained in Chapter 7, the appropriate key A to U is pressed. To exit from the graphics mode you must press the **GRAPHICS** key again; this re-establishes the **L** cursor.

To help you to familiarise yourself with operating the keyboard we include below a step-by-step description of how to enter a simple three-line Basic program, which when run calculates and displays the square root of an entered number.

Start off by pressing the **NEW** key when the **K** mode is selected and then **ENTER**, which clears the memory and brings up the Sinclair caption on the screen. Then carry out the following steps and note the display after each step.

Your action	*TV display after your action*	
	lower left	*upper left*
press 5	5 **K**	none
enter keyword **INPUT**	5 INPUT **L**	none
press y	5 INPUT y **L**	none
press **ENTER**	**K**	5>INPUT y
enter 15	15 **K**	5>INPUT y
enter keyword **PRINT**	15 PRINT **L**	5>INPUT y
enter extended mode (**CAPS SHIFT** and **SYMBOL SHIFT**)	15 PRINT **E**	5>INPUT y
enter **SQR** (H key)	15 PRINT SQR **L**	5>INPUT y
press y	15 PRINT SQR y **L**	5>INPUT y
press **ENTER**	**K**	5 INPUT y 15>PRINT SQR y
enter 25	25 **K**	5 INPUT y 15>PRINT SQR y

enter keyword **STOP** (**SYMBOL SHIFT** and A)	25 STOP L	5>INPUT y 15>PRINT SQR y
press **ENTER**	K	5 INPUT y 15 PRINT SQR y 25>STOP

Remarks or comments can be included in a program by using the **REM** statement. The comments following the **REM** statement are transparent when the ZX Spectrum executes the program. You may wish to verify this by including the line

2 **REM** Square Root Program

in the program above.

You will note that as each line appears in the upper left-hand part of the screen the symbol > appears in the line. This is a pointer used in program editing. We shall discuss the editing features of the ZX Spectrum later in this chapter.

The program has now been entered and can be run. To do this press **RUN** followed by **ENTER.** The program listing is then cleared from the screen and the L cursor appears in the lower left-hand corner prompting you to enter the number whose square root is required. You should now input the number and follow this by **ENTER**. For example, if you input 25 the result 5 will be displayed in the upper left-hand corner of the screen, and the report

9 **STOP** statement, 25:1

will be displayed in the lower part of the screen. This indicates that the program terminated with a **STOP** statement in line 25.

To re-run the program you have to input **RUN** followed by **ENTER**. If you wish to run the program for several different numbers it may be more appropriate to change line 25 to

25 **GO TO** 5

and in this case the program does not stop after displaying the result, but is directed to go back to line 5. The program is now in a continuous loop and will calculate and display the result of 22 entered numbers. If you input another number the display *scrolls*, ie the display moves up one line and the top line is removed.

When you are in the loop and wish to stop execution of the program, enter **STOP** (**SYMBOL SHIFT** and A) followed by **ENTER**, and in the simple program above the report

H STOP in INPUT, 5:1

will be displayed in the lower part of the screen. To continue the program after a stop action simply enter **CONT** followed by **ENTER.**

As a further example, consider the following one-line program to illustrate use of the graphics mode and the syntax monitoring feature

of the ZX Spectrum. When execution of the above program has been stopped, bring up the Sinclair caption by entering **NEW** as described in the previous example and then carry out the following steps, noting the display after each step.

Your action	*TV display after your action*	
	lower left	*upper left*
press 5	5 K	none
enter keyword **PRINT**	5 PRINT L	none
enter " (**SYMBOL SHIFT** and P)	5 PRINT " L	none
enter **GRAPHICS** mode (**CAPS SHIFT** and 9)	5 PRINT " G	none
enter ■■■ (key 6)	5 PRINT " ■■■ G	none
exit from **GRAPHICS** mode (**CAPS SHIFT** and 9)	5 PRINT " ■■■ L	none
press **ENTER**	5 PRINT " ■■■ ? L	none

The ? cursor indicates a syntax error because we have omitted the necessary delimiter, ", which must be included at the end of a **PRINT** statement. The correction is made as follows

Your action	*TV display after your action*	
	lower left	*upper left*
enter " (**SYMBOL SHIFT** and P)	5 PRINT "■■■" L	none
press **ENTER**	K	5>PRINT "■■■"

To run the one-line program press **RUN** followed by **ENTER.** You will now see ■■■ printed in the top left-hand corner of the screen, and the report

0 OK, 5:1

is displayed in the lower part of the screen, indicating successful completion of the program, and program termination at line 5.

When you *hold* a single key pressed (non-shifted or shifted) the character corresponding to the pressed key is assigned to the string variable **INKEY$**; otherwise **INKEY$** is a null string (""). You can verify this by entering and running the following program

```
 5 IF INKEY$ = "" THEN GO TO 5
15 PRINT INKEY$;
25 GO TO 5
```

The operation of this continuous program loop may be terminated by holding down the **CAPS SHIFT** and **BREAK** keys, and the resulting displayed report is

L BREAK into program, 5:2

You may assign the string variable **INKEY$** to a string variable (a single alphabetic character followed by a $) using the **LET** statement.

For example, we can change the above program to the alternative form

```
 5 IF INKEY$ = "" THEN GO TO 5
15 LET K$ = INKEY$
25 PRINT K$;
35 GO TO 5
```

The form and use of strings and string variables are described in Chapter 4. If you wish to clear the screen display enter the statement **CLS.**

Program listing and editing

To list lines of a program on the TV display you can use the keyword **LIST.** If you simply input **LIST** and then press **ENTER** the ZX Spectrum displays the first 22 lines of your program, and displays

scroll?

at the bottom of the screen when your program has more than 22 lines. This occurs because the screen is full. To display the next 22 lines of your program press y for yes, and you will see that the display scrolls. This process can be continued until the last line of your program is displayed. Then the Spectrum displays the report code

0 OK, 0:1

in the bottom left-hand corner of the screen. If, of course, your program is 22 lines or less it is displayed completely by **LIST** and **ENTER.**

If you press n instead of y when

scroll?

is displayed, the report

D BREAK — CONT repeats, 0:1

is displayed and you can now proceed with other keying-in operations.

An alternative method of listing 22 lines of a program is to use the instruction

LIST line number

followed by **ENTER.** This displays the specified line number at the top of the screen and then the next 21 lines of program. Using this approach you can examine any block of 22 lines of your program.

When you wish to change a line in a program you can do this in one of two ways. The first method simply involves entering the new

line of program in the normal way, which overwrites the exisiting line of program when **ENTER** is pressed. The alternative method is to use the line editing facility. To change a program line using this method first display the line to be edited and subsequent lines in the block by using the instruction

LIST line number to be edited

If scroll? is displayed, hold down the **CAPS SHIFT** and the 1 key until the line to be edited appears in the lower part of the screen. Otherwise select the **EDIT** mode (**CAPS SHIFT** and 1) and the line to be edited will now appear in the lower part of the screen, with the **K** cursor appearing after the line number.

Right and left movement of the **K** or **L** cursor in the line to be edited is achieved by ⇩ (**CAPS SHIFT** and 8) or ⇧ (**CAPS SHIFT** and 5) respectively. The alphanumeric character or word immediately to the left of the cursor can be removed using the **DELETE** key (**CAPS SHIFT** and 0), and an alphanumeric character or word may be inserted at the position of the cursor by entering it from the keyboard. After editing the line it is entered, replacing the original line when **ENTER** is pressed.

It is worth noting that, to assist editing, the line cursor may be moved up or down in a listed block of program by using the ⇦ (**CAPS SHIFT** and 7) or ⇨ (**CAPS SHIFT** and 6) respectively. You should also note that the edited line of program appears in the displayed program listing with the program cursor > after the line number.

Saving programs on cassette tape

The ZX Spectrum stores programs and information in its volatile memory, which means that when you turn off the power supply the program and any data will be lost. However, when you wish to keep a program for use at some future time you can copy (save) it from the memory of your microcomputer onto cassette tape and, when you wish to run the program again, you copy (load) it from the cassette tape into the memory of the ZX Spectrum.

To transfer a program from your ZX Spectrum to a cassette tape, you must connect the MIC lead (it has 3.5 mm jack plugs at either end) between the MIC socket on the ZX Spectrum and the corresponding input on your cassette tape recorder. You must ensure that *one of the jack plugs connecting the cassette recorder and computer EAR sockets is removed.* Then, with the tape in the position where you wish to record the program, you input

SAVE "Name of Program"

and press the **ENTER** key. Note that the Name of Program may

consist of up to ten alphanumeric characters. The computer responds by displaying the instruction caption

Start tape, then press any key.

You must therefore follow this instruction by setting the tape recorder into its record mode and then pressing any key on the ZX Spectrum keyboard. If you are using a colour TV set you will see moving patterns of red and pale blue stripes, followed by moving patterns of blue and yellow stripes (light and dark stripes on a monochrome TV), and then these will stop and the report

Ø OK, Ø:1

will be displayed, indicating successful recording of your program. You may now stop the cassette recorder.

To check that the signal has been successfully recorded you can use the **VERIFY** command. To verify you must first *reconnect the EAR jack plug,* then rewind the cassette to the point where you started recording the program, input

VERIFY "Name of Program"

and start the cassette recorder playing. After displaying a border alternating between red and blue colours (dark and light shades on a monochrome TV), the display changes to patterns similar to those obtained when **SAVE**ing the program and then outputs the report

Ø OK, Ø:1

indicating correct verification. You should then stop the cassette recorder.

If the program is not **SAVE**d correctly the ZX Spectrum will output the report

R Tape Loading error, Ø:1

after the **VERIFY** operation. You should then check the connections, adjust the volume and tone settings on the cassette recorder and try the **SAVE** and **VERIFY** operations again.

To transfer a program from a cassette tape into your ZX Spectrum you must position the tape so that it is at the beginning of the program, ensure that the EAR socket on your ZX Spectrum is connected to the earphone socket on the cassette recorder, enter

LOAD "Name of Program"

and start the cassette recorder playing. Once the computer has loaded the program it responds with the report

Ø OK, Ø:1

The program name and the name of the other programs found on the

tape will be listed on the screen. The keyword **RUN** is entered to execute the program once it has been loaded.

When the computer is being **LOAD**ed with a program from tape, the existing program and variables in the computer are first deleted and then the new program and variables are loaded into the computer memory. However, sometimes you may wish to maintain part of a program or a full program existing in the computer memory, and load into the memory an existing program stored on tape, thus creating a new program in the computer memory. This can be achieved using the **MERGE** statement. When the **MERGE** operation is implemented, only those lines of program already existing in memory that have the same line number as the new program being loaded from tape are deleted. Similarly, only those variables in memory that conflict with the same variable names on tape are deleted. The **MERGE** operation is achieved by using the command in the form

MERGE "Name of Program"

and the tape is loaded in the normal way, as described above.

You may also **SAVE** number arrays, character arrays and bytes from memory, and also **LOAD** these three sorts of information from cassette tape back into the computer.

The command

SAVE "Name of Program" **DATA** array name ( )

is used to **SAVE** the data in array name ( ) and the data in the array may subsequently be **LOAD**ed into an existing array of the same name in a new program. Details of numeric and string arrays are given in Chapters 3 and 4 respectively.

To save bytes of memory use the command

SAVE "Name of Bytes" **CODE** address of first byte to be saved, number of bytes to be saved

and to load bytes of memory use the command

LOAD "Name of Bytes" **CODE** address of first byte to be loaded, number of bytes to be loaded

During the loading process the TV screen displays

Bytes: Name of Bytes

and, after successfully loading the bytes, displays the report

Ø OK, Ø:1

in the lower part of the screen.

The TV screen display is mapped into 6912 bytes of memory, ie 6144 bytes for the display file + 768 bytes for the attribute file, starting at address 16384, so if you use the command

SAVE "screen" **CODE** 16384, 6912

and **SAVE** the result on cassette tape, you will store a copy of the TV screen display. To **LOAD** this back into memory you may use the command

LOAD "screen" **CODE** 16384, 6912

The **CODE** 16384, 6912 statement is most useful for referring to the screen display and it can be implemented using **SCREEN$**. Consequently the command

SAVE "screen" **SCREEN$**

can be used to **SAVE** on cassette tape the content of the TV display, and

LOAD "screen" **SCREEN$**

can be used to **LOAD** from cassette tape the saved content of the TV display. Note that we have used 'screen' as the name of the 6912 bytes of screen memory, but we could have used any alphanumeric characters, to a maximum of 10 in number.

Reports

You will notice when you have run a program that a report is displayed at the bottom of the screen, and in previous sections of this chapter we have given the reports which occur after certain specific tasks.

Reports always appear when the ZX Spectrum finishes executing Basic commands, either because the program is complete or because an error has occurred. All reports have a single alphanumeric character and a brief message that indicates what has occurred, and the line and statement number where the program has stopped. The form of the report is

report code message, line number where program stopped:
statement number where program stopped

Statements in a program line may be identified by a statement number. Statement 1 is at the beginning of the line, statement 2 is after **THEN** or the first colon, etc. If a command is implemented the report lists the line number as 0. Table 1.1 gives a summary of reports and their respective codes.

Table 1.1. ZX Spectrum reports

Report Code	*Meaning*
Ø	OK
1	**NEXT** used without **FOR**
2	Variable not found
3	Subscript wrong
4	Out of memory
5	Out of screen
6	Number too large
7	**RETURN** used without **GO SUB**
8	End of file
9	**STOP** statement
A	Invalid argument
B	Integer out of range
C	Nonsense in Basic
D	**BREAK — CONT**inue repeats
E	Out of **DATA**
F	Invalid file name
G	No room for line
H	**STOP** in **INPUT**
I	**FOR** used without **NEXT**
J	Invalid input output device
K	Invalid colour
L	**BREAK** into program
M	RAMTOP invalid
N	Statement lost
O	Invalid stream
P	**FN** used without **DEF**
Q	Parameter error
R	Tape loading error

The exposed edge connector

The exposed edge connector on the ZX Spectrum contains circuit connections to the Z80A microprocessor and special control lines so that extra peripheral devices can be added to your Spectrum. For example, the ZX Printer and the ZX Microdrive have been designed to plug directly onto the edge connector.

As the edge connector contains the pin connections of the Z80A microcomputer, it is possible to design interface and control circuitry which can be used to make your ZX Spectrum control external

systems. This can be achieved using some of the commercially available interface devices or you may eventually wish to design and build your own. In Chapter 9 we consider some aspects of the input/output (I/O) capabilities of the ZX Spectrum, and we discuss the edge connector and the signals available.

The ZX Printer (Plate 1.2)

The Printer enables you to obtain a permanent record of the display on the TV screen, and it permits long programs and lists of results to be printed. The output from the Printer consists of 32 columns, the same as the output on the TV screen, with as many rows as you wish. You use the **LPRINT, LLIST** and **COPY** statements to obtain *hard copies*.

The **LPRINT** statement is used in the same way as the **PRINT** statement, differing only in that the output is to the printer instead of to the screen. For example, if you input

LPRINT "I like Computers"

followed by **ENTER**, your message, I like Computers, will be printed.

To print lines of your program on the ZX Printer use the statement **LLIST**. This is similar in operation to the **LIST** statement used for displaying a maximum of 22 lines of program on the TV screen, but the **LLIST** statement causes *all* lines of your program to be printed. The **LLIST** operation can be terminated by pressing the **CAPS SHIFT** and **BREAK** keys. To start printing from a specified line number of a program enter

LLIST line number

The **COPY** statement is used to obtain a hard copy of the entire screen display. This feature is useful for obtaining graphs, tables of data, histograms, etc.

You should note that it is always possible to stop the Printer by entering **BREAK** (**CAPS SHIFT** and **SPACE**).

Concluding remarks

In this chapter we have introduced the ZX Spectrum and considered the keyboard and its operation. We have also introduced simple programming, editing of programs, saving programs and data on cassette tape, operating system reports and some uses for the exposed edge connector. This material will be useful to you when studying subsequent chapters of this book, and it has provided you with some basic 'hands-on' experience.

2

Binary and hexadecimal numbers

Introduction

We use decimal numbers in our everyday lives and accept and use them readily. Not surprisingly then, the data used with Basic programs is often in decimal form. The typical program statement

```
25 PRINT 17.4/6.29
```

uses the decimal line-number, 25, and the decimal data-numbers, 17.4 and 6.29. The ZX Spectrum outputs the answer 2.7662957 (decimal number) to the screen.

The electronic circuits inside the microcomputer are not designed to handle decimal numbers directly. In fact the decimal numbers used in a program statement are converted by the system into suitable equivalent forms which are acceptable to the electronic circuits. These equivalent forms are the binary representations of the numbers. Integer binary numbers are used, for example, when we include machine-code instructions in a program, or when we define the *attributes* of a displayed screen character.

Binary numbers

The ZX Spectrum uses integrated circuits (chips) which have two stable states. These are known as the binary states and they represent two voltage levels, say 0 volts and +5 volts. It is convenient to represent the 0 volt level by the symbol 0 and the +5 volt level by the symbol 1. The symbols 0 and 1 are called bits (binary digits).

Numbers can be represented by an arrangement of bits. This is possible because a numeric weighting is given to each bit position within the number. For example, the binary number 11100010 is a shorthand way of writing

$(1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3)$
$+ (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$

$= 128 + 64 + 32 + 0 + 0 + 0 + 2 + 0$

$= 226$

Exercise
Use this method to verify that

10101010 = 170

You may find it more convenient to use the **BIN** pseudo-function to convert this binary number to decimal using

PRINT BIN 10101010

Note that the **BIN** pseudo-function defines the number as being in binary form, but remember that **BIN** can only handle integer binary numbers having a maximum *word length* of sixteen bits. Should you wish to convert an integer binary number containing more than sixteen bits to its equivalent decimal form you can use the generalised approach described below.

An i-bit integer binary number partitioned into j-bits (most significant group) and k-bits (least significant group), where $0 \leqslant i \leqslant 32$, $0 \leqslant j \leqslant 16$ and $0 \leqslant k \leqslant 16$, has an equivalent decimal value given by

($2^k \times$ decimal value of j-bit word) + decimal value of k-bit word

In Basic this can be evaluated using

PRINT 2↑k***BIN** j-bit word + **BIN** k-bit word

For example, consider 11011110111101110010110 (23-bit word) partitioned as

$$\underbrace{11011110111101}_{j = 14 \text{ bits}} \quad \underbrace{110010110}_{k = 9 \text{ bits}}$$

This is converted to decimal using

PRINT 2↑9***BIN**11011110111101 + **BIN**110010110

and yields 7306134 as the displayed answer.

Exercise
Verify that the above 23-bit word partitioned as

$$\underbrace{1101111011}_{j = 10 \text{ bits}} \quad \underbrace{1101110010110}_{k = 13 \text{ bits}}$$

yields the same decimal answer (7306134) using

PRINT 2↑13***BIN**1101111011 + **BIN**1101110010110

A binary word of eight bits is called a *byte*. A single byte can represent any integer number in the range zero (00000000) to 255 (11111111). Alternatively, a byte in the ZX Spectrum can be used to represent a character in the Character Set. For example, the byte 00100100 (code 36) represents the $ character (see Table 7.1).

A sixteen-bit binary number consisting of two bytes, the least significant byte (LSB) and the most significant byte (MSB), can represent any integer in the range

$$\underbrace{00000000}_{\text{MSB}}\ \underbrace{00000000}_{\text{LSB}} \text{ to } \underbrace{11111111}_{\text{MSB}}\ \underbrace{11111111}_{\text{LSB}}$$

ie 0 to 65535.

Binary equivalent of a decimal number

The decimal number to be converted is first separated into its integer and fraction parts and the two parts are converted separately. After conversion the two parts are combined to yield the result.

The binary equivalent of the integer part of the decimal number can be obtained by repeatedly dividing the decimal number by 2 to form a quotient and a remainder. The division process is repeated until the quotient is zero, and the remainders formed by the conversion represent the number in binary form. The last remainder is the most significant bit of the binary number. The following example illustrates the method.

		Quotient	Remainder
(Start)	226 ÷ 2 =	113	0
	113 ÷ 2 =	56	1
	56 ÷ 2 =	28	0
	28 ÷ 2 =	14	0
	14 ÷ 2 =	7	0
	7 ÷ 2 =	3	1
	3 ÷ 2 =	1	1
(Finish)	1 ÷ 2 =	0	1

Read upwards

ie 226 (decimal) = 11100010 (binary)

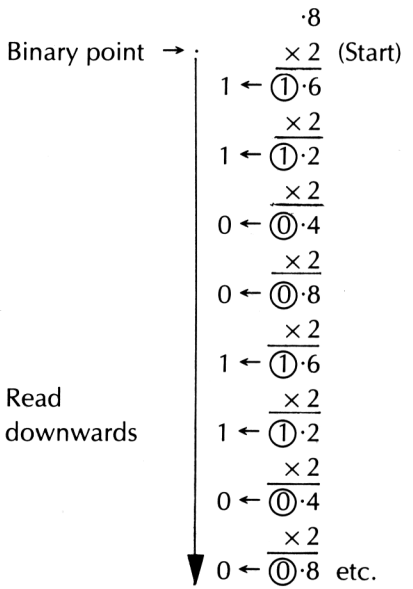
Exercise

Use this method to verify that

174 = 10101110
240 = 11110000

Program 1 in Appendix A can be used to convert an integer decimal number to its equivalent binary number.

The fraction part of the decimal number is converted by repeatedly multiplying the decimal fraction by 2 and noting the integer number generated. This process is repeated until sufficient accuracy has been obtained – it may not be possible to obtain an exact equivalent. The following example illustrates the method.



ie .8 (decimal) ≅ .11001100 (binary)

Exercises

1. Use this method to verify that

   .140625 = .0010010
   .9013671875 = .1110011011

2. Show that the binary equivalent of the decimal number 25.34375 is 11001.01011

Decimal equivalent of a binary number

The binary number to be converted is first separated into its integer and fraction parts and the two parts are converted separately. After conversion the two parts are combined to yield the result.

The decimal equivalent of the integer part of the binary number is obtained by multiplying the most significant bit of the binary number by 2 and the next significant bit is added to the result of the product. The result is then multiplied by 2. The next significant bit is now added to the result, and this is multiplied by 2. We continue in this way until all bits of the binary number have been used. The following example illustrates the method.

	1	0	0	1	0	0	1	1	
(Start)	× 2								
	2	+ 0							
		2							
		× 2							
		4	+ 0						
			4						
			× 2						
			8	+ 1					
				9					
				× 2					
				18	+ 0				
					18				
					× 2				
					36	+ 0			
						36			
						× 2			
						72	+ 1		
							73		
							× 2		
							146	+ 1	
								147	(Finish)

ie 10010011 (binary) = 147 (decimal)

Program 1 in Appendix A can be used to convert an integer binary number to its equivalent decimal form.

Exercise
Use this method to verify that

101101 = 45
11011000 = 216

The decimal equivalent of the fraction part of the binary number is obtained by dividing the least significant bit of the binary number by 2 and the next least significant bit is added to the result of the division. The process is continued until the most significant bit of the binary fractional number has been added and the resultant divided by 2.

The following example illustrates the method.

```
Binary                                          Decimal
point → .                                       point
                                                  ↓
        0                          2)0.8125 = .40625 (Finish)
        1                  2)1.625 = 0.8125
        1          2)1.25 = 0.625
        0  2)0.5 = 0.25
```

(Start) 2)1 = 0.5

ie .01101 (binary) = .40625 (decimal)

Exercises

1. Use this method to verify that

.11011 = .84375
.101001 = .640625

2. Show that the decimal equivalent of the binary number 1011001.100111 is 89.609375.

Hexadecimal numbers

Hexadecimal numbers are a convenient and very useful shorthand representation of their equivalent binary form. This representation uses the numeric symbols 0 to 9 and the alphabetic symbols A to F, as shown in Table 2.1.

We can see from Table 2.1 that an eight-bit binary number is represented by two hexadecimal symbols. The least significant hexadecimal symbol represents the least significant four bits of the binary word, and the most significant hexadecimal symbol represents the most significant four bits of the binary word. A numeric weighting is given to each hexadecimal symbol; the least significant symbol has a weighting of 1 (16^0) and the most significant symbol has a weighting of 16 (16^1). For example, B7 (10110111) may be written as

$$(B \times 16^1) + (7 \times 16^0)$$
$$= (11 \times 16) + (7 \times 1)$$
$$= 183$$

Exercise

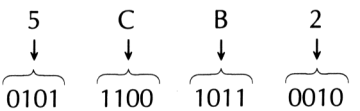
Using this method show that

E6 = 230

Table 2.1. Code conversion table

Decimal	Hexadecimal	Binary
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000
9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111
16	10	00010000
17	11	00010001
:	:	:
254	FE	11111110
255	FF	11111111

Note that hexadecimal numbers are not restricted to using just two weighted symbols. A typical example of using four weighted hexadecimal symbols in the ZX Spectrum is when a sixteen-bit word (two bytes) is used for the address code of a memory location. For example, the hexadecimal number 5CB2 written in binary form is



ie 5CB2 (hex) = 0101110010110010 (binary)

Exercise
Using this method show that

C3 = 11000011
BAD1 = 1011101011010001

When converting an integer hexadecimal number to its equivalent decimal form it may be appropriate to convert the hexadecimal

number to its equivalent binary form and then use the **BIN** pseudo-function to convert from binary to decimal as described earlier.

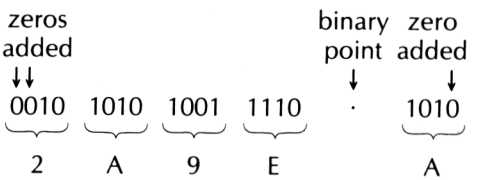
Hexadecimal equivalent of a binary number

A binary number can be converted directly to its hexadecimal equivalent by starting at the binary point and splitting the integer and fractional part of the number into groups of four bits. Zeros are added if necessary to complete a group of four. Each group is then converted directly to the appropriate single hexadecimal character shown in Table 2.2.

Table 2.2. Binary code to hexadecimal conversion

Binary group code	Hexadecimal character
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

The following example illustrates the method.



ie 10101010011110.101 (binary) = 2A9E.A (hex)

Exercise
Use this method to show that

11001.110 = 19.C
111010111100001 = 75E1

Hexadecimal equivalent of a decimal number

The decimal number to be converted is first separated into its integer and fraction parts and these two parts are converted separately. After conversion the two parts are combined to yield the result.

The hexadecimal equivalent of the integer part of a decimal number can be obtained by repeatedly dividing the decimal number by 16 to form a quotient and a remainder. The division process is repeated until the quotient is zero, and the remainders formed by the conversion represent the number in hexadecimal form. The last remainder is the most significant character of the hexadecimal number. The following example illustrates the method.

		Quotient	Remainder	
(Start)	23730 ÷ 16 =	1483	2	↑
	1483 ÷ 16 =	92	11 (B)	|
	92 ÷ 16 =	5	12 (C)	|
(Finish)	5 ÷ 16 =	0	5	|
				Read upwards

ie 23730 (decimal) = 5CB2 (hex)

Exercise
Using this method show that

212 = D4
19132 = 4ABC

Program 1 in Appendix A can be used to convert an integer decimal number to its equivalent hexadecimal number.

The fraction part of the decimal number is converted by repeatedly multiplying the decimal fraction by 16 and noting the integer number generated (which is replaced by its hexadecimal equivalent character). This process is repeated until sufficient accuracy has been obtained - it may not be possible to obtain an exact equivalent. The following example illustrates the method.

Hexadecimal point →	.	.8
		× 16 (Start)
C ← replace by ←		⑫.8
	hexadecimal equivalent	× 16
C ←		⑫.8
		× 16
C ←		⑫.8
	ad infinitum	etc.

ie .8 (decimal) ≅ .CCC (hex)

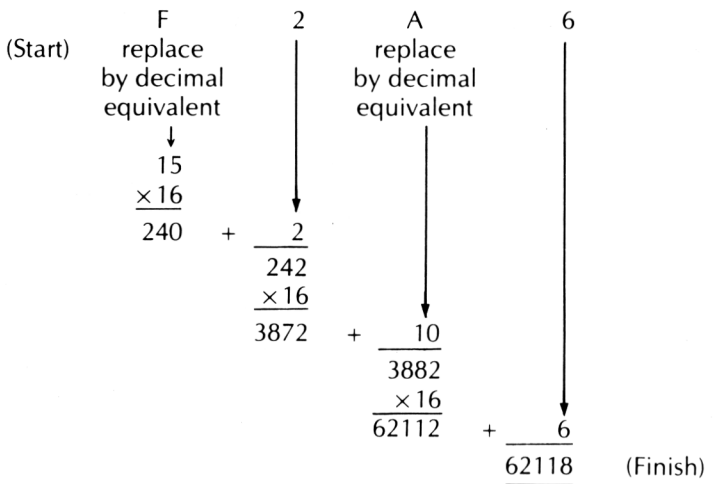
Exercises

1. Use this method to verify that
   .890625 = .E4
   .9013671875 = .E6C
2. Show that the hexadecimal equivalent of the decimal number 33.84375 is 21.D8

Decimal equivalent of a hexadecimal number

The hexadecimal number to be converted is first separated into its integer and fraction parts and these are converted separately. After conversion they are combined to give the final result.

The decimal equivalent of the integer part of the hexadecimal number is obtained by converting the most significant character of the hexadecimal number to its decimal equivalent and multiplying this by 16, and then the next significant character is converted to its decimal equivalent and added to the result of the product. The result is then multiplied by 16. The next significant character is converted to its decimal equivalent and added, and the result is then multiplied by 16. Continue in this way until all characters of the hexadecimal number have been used. The following example illustrates the method.

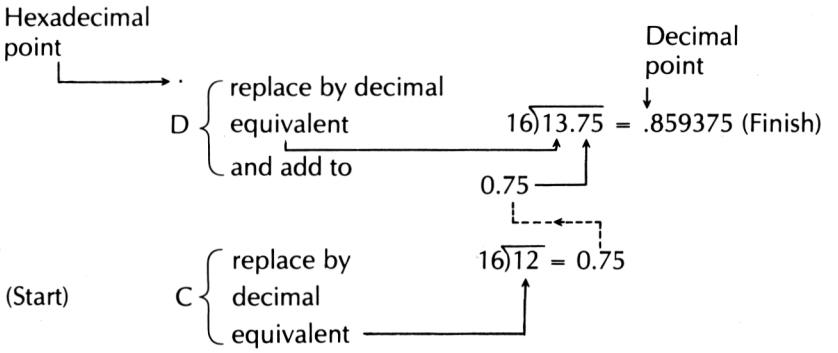


ie F2A6 (hex) = 62118 (decimal)

Program 1 in Appendix A can be used to convert an integer hexadecimal number to its equivalent decimal form.

The decimal equivalent of the fraction part of the hexadecimal number is obtained by converting the least significant character of the hexadecimal number to its decimal equivalent and then dividing

this by 16. The next least significant hexadecimal character is converted to its decimal equivalent and added to the result of the division. The procedure is continued until the most significant character of the hexadecimal fractional number has been converted to its decimal equivalent and added, and the result is divided by 16. The following example shows the method.



ie .DC (hex) = .859375 (decimal)

The integer and fraction parts are combined to give the result:

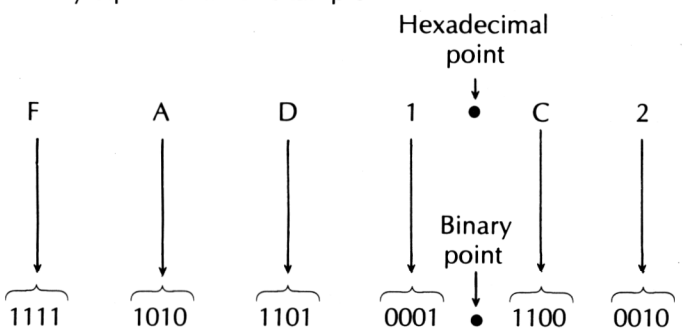
F2A6.DC = 62118.859375

Exercise

Show that the decimal equivalent of the hexadecimal number B2.F is 178.9375

Binary equivalent of a hexadecimal number

We have already seen in Table 2.2 that the hexadecimal form of a number is a shorthand representation of its binary equivalent. Consequently, to convert a hexadecimal number to its binary equivalent each hexadecimal character in the number is replaced by its binary equivalent. For example



ie FAD1.C2 (hex) = 1111101011010001.1100001 (binary)

Exercise
Show that the binary equivalent of E49.D1 is
111001001001.11010001

Numbers stored in memory

The Z80A microprocessor uses a 16-bit binary word to address a location in memory. However, it is more convenient to use the decimal equivalent of the 16-bit address word, and so the Basic interpreter in the ZX Spectrum allows you to refer to a memory address using a decimal number.

To examine the stored content of any memory location we use the **PEEK** function and the address of the memory location in decimal form. For example

PRINT PEEK 1983

will output to the screen the content of the read only memory (ROM) location 1983 which has the fixed stored value 58. Try this for yourself to check the result.

Exercise
Use the following program to display the content of a selected block of ROM, which is memory mapped in the decimal address range 0 to 16383. If you select a block of more than 22 locations you can scroll the display by pressing any key other than **N, SPACE** or **STOP,** because these cause a **BREAK.**

```
 2 PRINT "Input start address" : INPUT start:
   PRINT start
 3 PRINT "Input finish address" : INPUT finish:
   PRINT finish
 5 PAUSE 50
 7 CLS
 8 IF finish < start THEN PRINT "finish < start is not valid" :
   GO TO 2
10 FOR n = start TO finish
20 PRINT n,
25 PRINT PEEK n
30 NEXT n
```

Note that you can use Program 4 in Appendix A to obtain a hexadecimal listing of ROM.

It is only possible to change the content of a random access memory (RAM) location. We can do this by using the **POKE**

statement with the address of the memory location and the data in decimal form. For example

POKE 17111, 129

will load the eight-bit binary data word 10000001(129) into the display memory location 0100001011010111(17111). The corresponding eight-bit pattern is displayed at the addressed screen location (see Chapter 7 for more details of graphics). Do you now see why decimal number representation is more convenient than the equivalent binary representation? You can of course check that the data has been stored by **PEEK**ing this memory location with

PRINT PEEK 17111

You can store positive integer numbers in the range 0 to 255 using the **POKE** statement. If you attempt to store integer numbers greater than 255 the computer responds with the message: B Integer out of range.

You should note that if you **POKE** negative integer values in the range -1 to -255 they are accepted and stored in complement form. This means, for example, that if you **POKE** -123 it is actually stored as 133. Note that 133 = 256–123, which implies that the stored integer value is obtained by subtracting the magnitude of the entered negative number from 256.

If you try to store a non-integer number it is rounded to the nearest integer value and then stored. For example, try **POKE**ing 4.73 into memory location 17111. You may **PEEK** this location and see that the number is stored as 5. If you try to **POKE** 5.5 you will find that it is stored as 6, whereas 5.4999 is stored as 5. A negative non-integer number is rounded and stored in complement form. For example -13.417 is stored as 243 (ie 256–13).

Binary arithmetic

The Z80A microprocessor in the ZX Spectrum performs arithmetic operations using binary representations of numbers, and consequently, at the machine code level, a basic understanding of the concepts of binary arithmetic is desirable. To perform binary arithmetic operations it is simply a matter of knowing and applying the basic simple rules for addition, subtraction, multiplication and division.

Addition

The basic rules of binary addition are

0+0=0
0+1=1

1 + 0 = 1
1 + 1 = 0 with a carry 1
1 + 1 + 1 = 1 with a carry 1

Two examples of the addition of two four-bit binary numbers, with their equivalent decimal values, are shown below.

Decimal	Binary	Decimal	Binary
8	1000	7	0111
+7	+0111	+5	+0101
15	1111	12	1100

In these two simple examples we see that the addition of two four-bit binary numbers produces a four-bit result. However, if the sum exceeds 15 then the result is a five-bit word. For example

Decimal	Binary
9	1001
+8	+1000
17	10001

This shows that the addition of two n-bit words requires n + 1 bits to represent the result.

Exercise
Using the rules of binary addition show that 1101 + 0110 = 10011.

Subtraction

Binary subtraction may be performed using the basic binary subtraction rules, which are

0 − 0 = 0
1 − 0 = 1
1 − 1 = 0
0 − 1 = 1 with a borrow 1
1 − 1 − 1 = 1 with a borrow 1

Two examples of the subtraction of two four-bit binary numbers, with their equivalent decimal values, are shown below.

Decimal	Binary	Decimal	Binary
14	1110	12	1100
−3	0011	−7	0111
11	1011	5	0101

In these two subtraction examples the results are positive numbers, but the result of a subtraction may be positive or negative depending

on the relative magnitudes of the minuend and subtrahend. In order to distinguish whether a number is positive or negative the leftmost bit (most significant bit, MSB) is used as a *sign bit*. A commonly used convention is that the sign bit is 1 if the number is negative and 0 if the number is positive. Consequently the Z80A microprocessor, which has eight bits available for data, uses seven bits for the magnitude of the data and the most significant bit to indicate the sign.

Complex electronic circuitry is required to subtract binary numbers directly, and in microcomputers it is usual to perform subtractions by adding the two's complement of the subtrahend to the minuend. This means that subtractor circuitry is not required and subtraction is performed using adder circuits to implement the complement form of arithmetic operations.

The two's complement of a binary number is found by inverting each bit of the binary number, that is, changing all the 0s to 1s and 1s to 0s, and adding 1 to the result. For example, the two's complement of 76 using an eight-bit binary representation is obtained as follows:

```
          sign bit
          ↓
+ 76 = 01001100
       ↓↓↓↓↓↓↓↓   change 1s to 0s and 0s to 1s
       10110011
            +1    add 1
       ________
       10110100 = −76 in two's complement form
```

To perform subtraction using the two's complement method, the two's complement of the subtrahend must be obtained, and this is then added to the minuend. As an example let us consider subtracting the decimal number 44 from decimal number 15 using eight-bit binary words.

```
     sign bit          sign bit
     ↓                 ↓
15 = 00001111,   44 = 00101100
                                  sign bit
                                  ↓
```

hence −44 in two's complement form is 11010100,

```
  sign bit
  ↓
  0 0001111
+ 1 1010100
  _________
  1 1100011  = −29 in two's complement form; that is, it is
               the two's complement of +29.
```

In this case the sign bit indicates a negative number which is in two's complement form.

Exercise

Let A and B be two's complement six-bit binary numbers, where A = 010111 and B = 010001. Show that A–B = 000110 and B–A = 111010.

As a further example of the two's complement method of subtraction we shall consider subtracting the decimal fraction 0.25 from the decimal fraction 0.875 using eight-bit binary words. In this case we must first convert both decimal fractions to their equivalent binary representations. This is done as follows:

$$0.875 = 0.5 + 0.25 + 0.125$$
$$= (1 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

In shorthand notation we write

$$0.875 = 0.111$$

Similarly, $0.25 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3})$

ie $0.25 = 0.010$

Using eight-bit binary words for the binary numbers, with the binary point occurring after the fourth bit, we have

sign bit ↓ binary point ↓ sign bit ↓ binary point ↓

$0.875 = 0000.1110, \quad 0.25 = 0000.0100$

sign bit ↓ binary point ↓

hence –0.25 in two's complement form is 1111.1100

sign bit ↓ binary point ↓

0 000.1110 (= 0.875)
+ 1 111.1100 (= –0.25)

ignore overflow → 1 0 000.1010 (= 0.625)

Exercises

1. Using eight-bit binary words, with five bits for the binary fraction, show that the two's complement representation of 0.53125 = 000.10001 and –0.6875 = 111.01010.

2. Verify that

$$\begin{array}{rl} & 000.10001\ (=\ 0.53125) \\ + & \underline{111.01010}\ (=\ -0.6875) \\ & \underline{111.11011}\ (=\ -0.15625) \end{array}$$

Multiplication

A common method of binary multiplication uses the shift and add principle. The multiplicand is multiplied by each bit of the multiplier on a bit-by-bit basis and the resulting product is obtained by adding all the appropriately shifted partial products. As the multiplier bits are either 0 or 1 the partial product terms are either zero or equal to the multiplicand or shifted versions of the multiplicand.

Let us illustrate the method by considering the multiplication of the decimal numbers 193 and 21. We use a 16-bit accumulator to hold the result and work with eight-bit binary number representations for the multiplicand and the multiplier.

Multiplicand	11000001 = 193	
Multiplier	00010101 = 21	
	11000001	Appropriately shifted partial product terms
	11000001	
	11000001	
	111111010101	Sum of partial product terms = 4053

Exercise

Using the method above show that $1101 \times 10111 = 100101011$

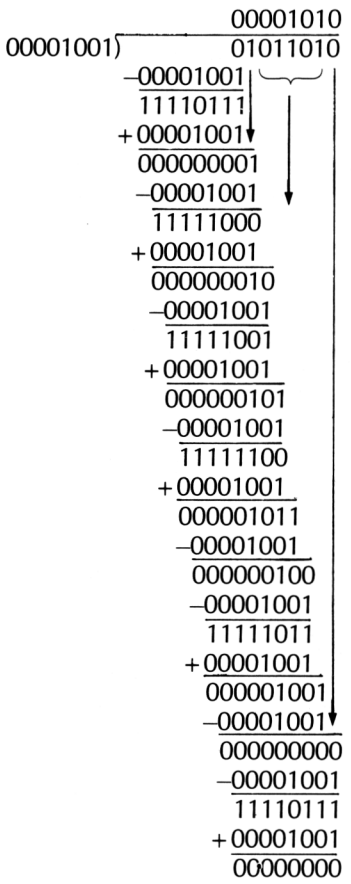
When two binary numbers are multiplied the product contains a number of bits equal to the sum of the bits contained in the two binary numbers. The maximum number of additions required for a full multiplication using the shift and add method is equal to the number of bits in the multiplier.

Division

Binary division is performed by using a shift and subtract method. It is implemented by successively subtracting the divisor from the appropriate shifted dividend, and inspecting the sign of the remainder after each subtraction. If the sign of the remainder is

positive the value for the quotient is 1, but if the sign is negative the value is 0, and the dividend is restored to its previous value by adding back the divisor (restoring). After the subtraction yielding a positive quotient, or after the restoration following a negative quotient, the divisor is shifted one place to the right and the next significant bit of the dividend is included and the operation repeated until all bits in the dividend have been used.

We illustrate the method by considering the division of 90 by 9 using corresponding eight-bit binary number representations.



00001010	
00001001) 01011010	
−00001001	
11110111	Result is negative therefore quotient value is 0
+00001001	Add divisor back
000000001	
−00001001	Shift divisor right and subtract
11111000	Result is negative therefore quotient value is 0
+00001001	Add divisor back
000000010	
−00001001	Shift divisor right and subtract
11111001	Result is negative therefore quotient value is 0
+00001001	Add divisor back
000000101	
−00001001	Shift divisor right and subtract
11111100	Result is negative therefore quotient value is 0
+00001001	Add divisor back
000001011	
−00001001	Shift divisor right and subtract
000000100	Result is positive therefore quotient value is 1
−00001001	Shift divisor right and subtract
11111011	Result is negative therefore quotient value is 0
+00001001	Add divisor back
000001001	
−00001001	Shift divisor right and subtract
000000000	Result is positive therefore quotient value is 1
−00001001	Shift divisor right and subtract
11110111	Result is negative therefore quotient value is 0
+00001001	Add divisor back
00000000	Remainder

Exercise

Using the method above show that 01011 ÷ 011 gives the quotient 0011 with the remainder 010.

Concluding remarks

In this chapter you have seen numbers represented in decimal, binary and hexadecimal forms and learnt how to convert from one form to another and how to **PEEK** and **POKE**. You have also been introduced to binary addition, subtraction, multiplication and division. You will find that a good working knowledge of these topics will assist you when programming and using your ZX Spectrum.

3

Number crunching

Introduction

An important feature of the ZX Spectrum is its ability to manipulate numbers and perform mathematical operations. This chapter examines how the computer represents numbers in floating point form and explains how arithmetic calculations are evaluated. We also describe the available mathematical and trigonometrical functions and show how to use the user-defined function capability. The chapter also includes an explanation of numeric variable arrays but string arrays will be dealt with in Chapter 4.

Software (programs) for video games may include routines which use random numbers, and we describe how these may be generated.

Floating point number representation

When using Basic in the ZX Spectrum, decimal numbers are represented using floating point binary notation. This permits manipulation of decimal numbers in the range $\pm 1.7 * 10^{38}$ using at least eight significant decimal digits.

In floating point form decimal numbers are represented as

$$\text{Number} = m * 2^{\varepsilon}$$

where ε is the exponent and m is the mantissa. The range of the mantissa is restricted to

$$\tfrac{1}{2} \leqslant m < 1$$

and the exponent ε is an integer in the range

$$-127 \leqslant \varepsilon \leqslant +127$$

The mantissa is stored as a four-byte fractional binary number, and because of its restricted range the most significant bit is always 1. This means that the most significant bit of the mantissa is redundant and it is therefore possible to use it to indicate the sign of the number. The convention adopted for this sign bit is to use 0 for positive numbers and 1 for negative numbers.

The exponent is stored using one byte. To avoid unnecessary complication regarding the sign of the exponent it is stored as a positive integer number in the range 1 to 255, achieved by adding the decimal number 128 to the actual exponent value, ε.

As an example we illustrate how the decimal number –28.84375 is stored using five bytes of memory. Consider

$$\begin{aligned} -28.84375 &= -0.9013671875 \times 32 \\ &= \underbrace{-0.9013671875}_{m} \times 2^{\overbrace{5}^{\varepsilon}} \end{aligned}$$

The fractional binary number representation of the magnitude of the mantissa, m, is .1110011011. However, the most significant bit of this number is always 1 and because it is redundant it is replaced by the sign bit. In this example since the mantissa is negative the sign bit is 1 and using four bytes for the sign and magnitude of the mantissa yields

11100110 11000000 00000000 00000000
↑
sign bit

The exponent value is 5 and 128 is added to it to yield 133, which is stored as the single-byte positive integer 10000101.

The ZX Spectrum represents zero by setting the four bytes of the mantissa, and the exponent byte, to 0.

Exercise
Show that the decimal number 140 is stored in floating point form as

sign bit
↓
10001000 00001100 00000000 00000000 00000000
$\varepsilon + 128$ m

Using the floating point format you will see that the maximum positive number that can be handled by the ZX Spectrum corresponds to when all bits in the exponent byte are set to 1,

representing $\varepsilon = +127$, and all 32 bits in the four bytes of the mantissa are also set to 1, representing $m = (1-(1/2^{32}))$. The maximum possible number is therefore

$$= 2^{127} \times (1-(1/2^{32}))$$
$$\cong 2^{127} \text{ because } (1-1/2^{32}) \cong 1$$
$$\cong 1.7014 \times 10^{38}$$

You can check that the ZX Spectrum can handle this number by entering **PRINT** 1.7Ø14E+38 in which case the computer responds by displaying 1.7Ø14E+38, indicating that the number is acceptable. However, if we increase this number to, say, 1.7015×10^{38}, which we enter as **PRINT** 1.7Ø15E+38, the computer will not accept the number because it is out of range.

The smallest positive number that can be represented in the ZX Spectrum corresponds to when the exponent byte produces $\varepsilon = -127$, and the mantissa value is 0.5. The minimum possible number is therefore

$$= 2^{-127} \times 0.5$$
$$= 2^{-128}$$
$$\cong 2.9388 \times 10^{-39}$$

You can confirm that the ZX Spectrum can handle this number by entering **PRINT** 2.9388E−39, in which case the computer responds by displaying 2.9388E−39, indicating that the number is acceptable. If however we try to input a smaller number, say 2.92E−39, by inputting **PRINT** 2.92E−39, the computer will not accept it but outputs its smallest positive number 2.9387359E−39. You may observe that for numbers in the range 2.9387359E−39 to 1.469367939E−39 the ZX Spectrum outputs its smallest positive number 2.9387359E−39, and for the number 1.469367938E−39 or less it outputs 0.

The ZX Spectrum uses a sign and magnitude format for number representation which means that the positive and negative number ranges are identical.

Exercise
Use the program below to investigate the acceptable number range of the ZX Spectrum.

```
 5 PRINT "Input number"
15 INPUT X
25 PRINT "Number is ";X
35 PRINT
45 GO TO 5
```

Numeric variables

When we use the ZX Spectrum to obtain the sum of three numbers, say 27.6, 8.7 and 9.1, we could simply command the computer to

PRINT 27.6 + 8.7 + 9.1

and the computer would display 45.4.

On the other hand if we wish to program the computer to add any three numbers then we would assign three different symbols to represent the three numbers, and use these symbols in the Basic program. For example, a simple program to obtain the required sum using the symbols x, y and z for the three numbers is

```
10 INPUT x
20 INPUT y
30 INPUT z
40 PRINT x + y + z
```

The symbols x, y and z may have any valid numeric value and consequently they are called numeric variables. Note that the **INPUT** statements used in lines 10, 20 and 30 require you to enter data for x, y and z via the keyboard.

A numeric variable can be represented using alphabetic characters (upper and lower case), numeric characters and spaces, but the first character must be alphabetic. You cannot use alphabetic characters in lower and upper case form to represent two different numeric variables. The ZX Spectrum will treat them as a single numeric variable. Some numeric variables are

James
salmon *or* Salmon
U12 *or* u12
The Final dividend is

However, the following numeric variables are not allowed

4 Sale because it begins with a numeric character
U12? because it uses an invalid character (?)

The **LET** statement can be used within a program to assign a constant value to a variable; for example

LET Rate = 6

or it is used to assign the value of the right-hand side of a mathematical expression, involving already defined numeric variables, to a new numeric variable. Typical examples are

LET x = 2 * Y + B
LET SUM 1 = −(4.9/t + A)

Another method of assigning constants to variables within a program is to use the **READ** statement and the necessary **DATA** statement. We can illustrate this using the following simple program

```
15 DATA 6, -7, 2.9, 149.21
25 READ x, y
35 PRINT x + y
45 READ p, q
55 PRINT p*q
```

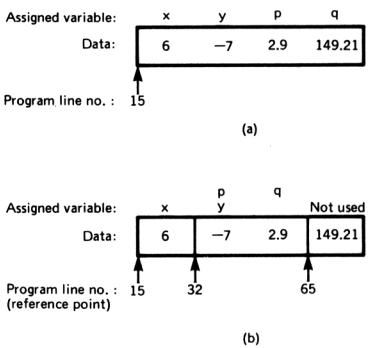


Fig. 3.1. Data list format: (a) Single reference point; (b) Three reference points

Note that line 15 sets up a data list, shown in Fig. 3.1(a), which is used by the **READ** statements. Line 25 assigns the first two data values to the variables x and y and, having used the first two data values, line 45 assigns the next two values in the data list to the variables p and q. Lines 35 and 55 are used to form the sum x + y and product p × q respectively.

The data values from one or more **DATA** statements are set up as a single data list and data is accessed in turn via **READ** statements. In the above program the data may be defined using more than one **DATA** statement; for example

```
15 DATA 6
25 READ x, y
32 DATA -7, 2.9
35 PRINT x + y
45 READ p, q
55 PRINT p*q
65 DATA 149.21
```

You can verify that this program gives the same numeric results as the previous program, −1 for the sum x + y and 432.709 for the product p × q. The merit of splitting up the data into several **DATA** statements is that it allows **READ** statements to access data from

specified reference points in the entire data list, as shown in Fig. 3.1(b). A reference point is set by the **RESTORE** statement followed by the program line number of the **DATA** statement from which data is to be accessed. You can examine this feature by inserting

```
40 RESTORE 32
```

in the above program. In this case you will see that the sum is −1, (same as before), but the product is now −20.3 because the **RESTORE** statement sets the data reference pointer to the data in line 32 of the program. In fact you could achieve the same results by restoring the reference pointer using any line number in the range 16 to 32, because this is interpreted by the ZX Spectrum as an instruction to access data from the next **DATA** statement at or following the line number specified in the **RESTORE** statement. To restore the reference pointer to the first data value in the data list you simply omit the line number in the **RESTORE** statement.

Exercises

1. Change line 40 in the above program to

```
40 RESTORE 18
```

and verify that the results are −1 and −20.3.

2. Change line 40 in the above program to

```
40 RESTORE
```

and verify that the results are −1 and −42.

When a program is saved the latest data entered or assigned to numeric variables is also recorded, so these are the initial data values assigned to the numeric variables immediately after the program is loaded. The **CLEAR** statement eliminates all defined variables and frees all of the memory that has been assigned to the variables.

Simple arithmetic calculations

The ZX Spectrum can directly perform five arithmetic operations, addition (+), subtraction (−), multiplication (*), division (/) and exponentiation (raising to a power) (↑). It is important to note that the computer cannot perform exponentiation on a negative number. When several arithmetic operations are used in the same calculation the computer firstly calculates any exponentiations, then any multiplications and divisions and then any additions and subtractions. However, if calculations are placed in parentheses they are evaluated first.

The best way to appreciate how calculations are worked out on the ZX Spectrum is to consider some examples. Try these for yourself.

1. Required calculation: $18.6 \times 3.7 - 2^{1.6}$
   ZX Spectrum statement: **PRINT** 18.6 * 3.7 − 2 ↑ 1.6
   Displayed answer: 65.788567
2. Required calculation: $18.6 \times (3.7 - 2^{1.6})$
   ZX Spectrum statement: **PRINT** 18.6 * (3.7 − 2 ↑ 1.6)
   Displayed answer: 12.435344
3. Required calculation: −0.06 ÷ 2.417 + 2.3E4
   ZX Spectrum statement: **PRINT** −0.06 / 2.417 + 2.3E4
   Displayed answer: 22999.975
4. Required calculation: $49^{-7.02}$
   ZX Spectrum expression: **LET** A = 49 ↑ −7.02 : **PRINT** A
   Displayed answer: 1.3640287E−12
5. Required calculation: $-10.66^{(4.2 \times 1.915)}$
   ZX Spectrum expression: **LET** Answer = −10.66 ↑ (4.2 * 1.915) : **PRINT** Answer
   Displayed answer: −1.8460829E + 8

Note that the displayed answers to examples 4 and 5 are given in scientific notation form, where the E−12 represents 10^{-12} and E + 8 represents 10^{+8}. This form of notation is a shorthand method of representing very large or very small numbers, and is sometimes referred to as floating point representation.

You can see from examples 4 and 5 that it is possible to include numeric variables in arithmetic expressions and the computer will then use their respective numeric values in the calculation. If we wish to write a program to calculate the average of any three numbers entered via the keyboard, then we can use the numeric variables a, b and c to represent the numbers, and calculate the average using the program listed below. Try it for yourself.

```
15 INPUT a : PRINT a
25 INPUT b : PRINT b
35 INPUT c : PRINT c
45 PRINT
55 PRINT "Average = "; (a + b + c)/3
```

As a further example, here is a program that will calculate both the square and the square root of a number entered via the keyboard.

```
15 INPUT A : PRINT A
20 PRINT
25 PRINT A↑2, A↑.5
30 PRINT
35 PRINT A↑2; A↑.5
40 PRINT
45 PRINT A↑2;" "; A↑.5
50 PRINT
55 PRINT A↑2'A↑.5
```

We suggest that you try this program and note the four different displayed answer formats created by the four different **PRINT** statements in lines 25, 35, 45 and 55.

Exercise
Use the computer as a calculator to show that

(i) $17.2^2 - 94.6 = 201.24$
(ii) $(8.3 \div 7.2 + 19.6)^{1/2} = 4.5555217$
(iii) $1/2 + 7/16 - 8.2^{0.6} = -2.5966771$

Mathematical functions

When using mathematical functions you should note that, in the absence of parentheses, these functions are evaluated before calculations in an expression involving the simple arithmetic operations: +, –, * and /. The rule that expressions enclosed in parentheses are evaluated first applies even when the total expression contains one or more mathematical functions.

ABS

The **ABS** function is used when you require the ABSolute value of either a number or an evaluated mathematical expression. The term 'absolute value' refers only to the magnitude of the number or evaluated expression, with the sign ignored. Use your ZX Spectrum to obtain the results of the following

```
15 PRINT ABS 118.97, ABS -118.97
25 PRINT ABS 2.9/1.12*2.81, ABS 2.9/-1.12*2.81
```

Did you notice that the fourth result is prefixed by the – sign? The reason for this is that, because the fourth expression does not contain parentheses, the ZX Spectrum first works out the absolute value of 2.9, then divides this by –1.12 and the result is multiplied by 2.81.

SGN

The **SGN** function is used to determine whether a number, or an evaluated mathematical expression, is zero, positive or negative. The result obtained when using this function is 0, +1 or −1 corresponding to the zero, positive and negative states respectively. A typical application of this function is where we wish the computer to accept only entered positive numbers greater than zero. This can be demonstrated using

```
 5 PRINT "Input Positive Number > Ø"
15 INPUT N
25 LET A = SGN N
35 IF A = Ø THEN GOTO 5
45 IF A = -1 THEN GOTO 5
55 PRINT N; "Enter Next Number"
65 GOTO 15
```

This program takes in a number and if it is positive displays it and requests you to enter your next number. A negative number or zero is not accepted and you are instructed to input a positive number greater than zero.

INT

The **INT** function is used to round down any number to the nearest integer value. A positive number is therefore reduced in magnitude by using this function, unless the number is already an integer, while a negative number is increased in magnitude unless it too is already an integer. For example, +17.632 is reduced to +17 but −17.632 is integerised to −18, that is its magnitude is increased. Here is a simple program which you can use to verify this for yourself.

```
 5 PRINT "Input a Number"
15 INPUT A
25 LET B = INT A
35 PRINT "The Integer of the Number is = ";B
45 GO TO 5
```

SQR

To obtain the square root of a number you can use the **SQR** function. Try the following program which allows you to find the square root of a positive number. You will note that we have used the **SGN** function to ensure that only positive numbers are accepted by the program.

This has been included in the program because the ZX Spectrum cannot calculate the square root of a negative number.

```
 5 PRINT "Input positive number >0": PRINT
15 INPUT n : PRINT n : PRINT
25 LET b = SGN n
35 IF b = Ø THEN GO TO 5
45 IF b = -1 THEN GO TO 5
55 LET c = SQR n
65 PRINT "The square root of "; n;" is "
68 PRINT : PRINT c
75 PRINT : GO TO 5
```

LN

To find the natural logarithm of a number you may use the **LN** function. The natural logarithm of a number, or $\log_e$ as it is sometimes called, is related to $\log_{10}$ by the relationship

$$\log_{10}x = \log_e x/2.3Ø2585Ø93$$

so we can use this to obtain $\log_{10}$ of a number. The program listed below uses the **LN** function and the above relationship to evaluate $\log_{10}$ of a positive number.

```
 5 PRINT "Input positive number >0": PRINT
15 INPUT n : PRINT n : PRINT
25 LET b = SGN n
35 IF b = Ø THEN GO TO 5
45 IF b = -1 THEN GO TO 5
55 LET x = LN n/2.3Ø2585Ø93
65 PRINT "Log 10 of "; n;" is"
68 PRINT : PRINT x
75 PRINT : GO TO 5
```

You may, of course, find the antilog_{10} of a number, x, by using the relationship

$$y = \text{antilog}_{10}\, x = 10^x$$

which can be implemented in Basic using

```
LET y = 1Ø↑x
```

EXP

For a number x we may evaluate the exponential function, e^x, using the **EXP** function. This relationship can be used, for example, to

evaluate exponential growth or decay. The program below illustrates the use of this function to evaluate the exponential growth of the mathematical term Ke^{2t}, for t = 0, 1, 2, . . . 15.

```
 3 INPUT K : PRINT K : PRINT
 5 FOR t = Ø TO 15
15 LET x = K*EXP(2*t)
25 PRINT x
35 NEXT t
```

We suggest that you run the program and then change line 15 to

```
15 LET x = K*EXP(–2*t)
```

Now re-run the program and observe the corresponding exponential decay.

Trigonometrical functions

When using trigonometrical functions note that, in the absence of parentheses, these functions are evaluated before calculations in an expression involving the simple arithmetic operations: +, –, * and /. The rule that expressions enclosed in parentheses are evaluated first applies even when the total expression contains one or more mathematical functions.

SIN, COS and TAN

The ZX Spectrum can evaluate the sine, cosine and tangent of an angle using the **SIN, COS** and **TAN** functions respectively. The angle must be expressed in radians, so if the angle is expressed in degrees use the relationship

Angle (in radians) = Angle (in degrees). $\pi/180$

The program below calculates the sine, cosine and tangent of an entered angle in degrees, where the mathematical constant π = **PI** ≅ 3.1415927.

```
15 PRINT "Input Angle in Degrees"
25 INPUT a : PRINT a : PRINT
35 LET x = a*PI/180
45 PRINT "Sin a = ", SIN x, "Cos a = ", COS x, "Tan a = ", TAN x
55 PRINT : GO TO 15
```

Try this program for various positive and negative angles. You will note that you cannot obtain the tangent of 90 degrees or 270 degrees because, of course, the value is infinite and outside the number range of the computer.

Exercise

Using the **SIN, COS** and **TAN** functions write BASIC statements to prove that

(i) $\tan A = \frac{\sin A}{\cos A}$

(ii) $\sin^2 a + \cos^2 a = 1$

(iii) $\sin 2x = 2\sin x \cos x$

(iv) $\cos 2x = \cos^2 x - \sin^2 x$

ASN, ACS and ATN

When you know the sine of the angle you can determine the corresponding angle using the arcsine function **ASN.** Similarly, if you know the cosine of the angle you find the angle by using the arccosine function **ACS**, and given the tangent of the angle you can find the angle using the arctangent function **ATN.**

These three functions give the resulting angle in radians and therefore to convert to degrees use the relationship

Angle (degrees) = Angle (radians).$180/\pi$

The following program calculates the angle in degrees for an entered sine value.

```
15 PRINT "Input Sine Value" : PRINT
25 INPUT s : PRINT s : PRINT
35 LET a = ASN s
45 LET b = a*180/PI
55 PRINT "Angle is "; b;" degrees"
65 PRINT : GO TO 15
```

When you run this program for positive and negative arcsine values in the valid range +1 to −1, you will note that it displays the calculated angle in the range +90 degrees to −90 degrees.

If you change line 15 to

```
15 PRINT "Input Cosine Value" : PRINT
```

and line 35 to

```
35 LET a = ACS s
```

the program above can be used to calculate the angle in degrees for an entered cosine value. The program displays the calculated angle in the range 0 degrees to 180 degrees corresponding to valid arccosine input values in the range +1 to −1.

If you change line 15 to

```
15 PRINT "Input Tangent Value" : PRINT
```

and line 35 to

```
35 LET a = ATN s
```

the program can be used to calculate the angle in degrees for an entered tangent value. In this case you will see that the program displays the calculated angle in the range +90 degrees to –90 degrees. Try the arctangent program for input values in the range +90000 to –90000.

User defined mathematical functions

It is possible to define your own functions for use within a Basic program. These are *user defined functions* and the format is

line number **DEF FN** letter (argument) = mathematical function

For example

```
35 DEF FN A(r) = PI*r↑2
```

defines the user function A(r) as being equal to πr^2 (the area of a circle having radius r). Having defined the function it may be used by referring to it as **FN** A(r), and it will be evaluated for any given argument. You may wish to prove for yourself that the command **PRINT FN** A(2) will evaluate the function and display the answer 12.566371, which is the area of a circle having radius (argument) 2.

Exercise
Enter and run the following program for various values of radius.

```
 5 PRINT "Input radius value"
15 INPUT r : PRINT r
25 PRINT
35 DEF FN A(r) = PI*r↑2
45 DEF FN V(r) = 4/3*PI*r↑3
55 PRINT "Area of circle with radius" 'r;" is "; FN A(r)
7Ø PRINT
75 PRINT "Volume of sphere with radius" 'r;" is "; FN V(r)
```

If you are wondering why the computer responds with A Invalid Argument, 55:1 if you input a negative radius value recall the earlier warning concerning exponentiation of negative numbers.

Another example where it may be appropriate to employ user

defined functions is in evaluating the hyperbolic functions: sinh x, cosh x and tanh x.

Using the series expansion form for sinh x

$$\sinh x \simeq x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots$$

we can define the function in Basic as

DEF FN s(x) = x + x↑3/6 + x↑5/120 + x↑7/5040

The series expansion form for cosh x is

$$\cosh x \simeq 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \ldots$$

and this may be defined in Basic as

DEF FN c(x) = 1 + x↑2/2 + x↑4/24 + x↑6/720

So we can now define tanh x as

DEF FN t(x) = **FN** s(x)/**FN** c(x)

The program listed below can be used to evaluate these hyperbolic functions for any positive value of x. Try running this program and compare the answers with the listed values given in standard mathematical tables, or evaluated using a suitable pocket calculator.

```
 35 DEF FN s(x) = x + x↑3/6 + x↑5/120 + x↑7/5040
 40 DEF FN c(x) = 1 + x↑2/2 + x↑4/24 + x↑6/720
 50 DEF FN t(x) = FN s(x)/FN c(x)
130 PRINT "Input value of x"
135 INPUT x
140 PRINT "For x = ";x' "sinh x = "; FN s(x) ' "cosh x = ";FN
    c(x)' " tanh x = ";FN t(x)
150 PRINT
155 PRINT "Enter G to GO AGAIN or S to STOP"
165 INPUT k$
175 IF k$ = "S" THEN STOP
185 IF k$ = "G" THEN GO TO 130
```

Note that if more precision is required in the calculated values of the hyperbolic functions, more terms must be used in the series expansions for sinh x and cosh x. For example, $x^9/9!$ and $x^8/8!$ are the next terms in the series which must be added to the definitions of sinh x and cosh x respectively to increase the series expansion of these hyperbolic functions by one additional term.

Exercise
Investigate the effect of adding + x↑19/362880 onto the end of line 35 in the above program; and + x↑18/40320 onto to the end of line 40 in the above program.

Alternative, and more precise, definitions for sinh x and cosh x are:

$$\sinh x = \frac{e^x - e^{-x}}{2} \quad \text{and} \quad \cosh x = \frac{e^x + e^{-x}}{2}$$

and these can be defined in the program above by changing lines 35 and 40 to

```
35 DEF FN s(x) = (EXP x - EXP - x)/2
40 DEF FN c(x) = (EXP x + EXP - x)/2
```

Exercise
For x = 0.8 and x = 3.1 use:
(i) the above series expansion program with (a) the original four terms and, (b) five terms (see previous Exercise); and
(ii) the above alternative definitions
to evaluate sinh x, cosh x and tanh x. Compare the results with the listed values given in standard mathematical tables, or evaluated using a suitable pocket calculator.

Numeric variable arrays

If several variables have one common characteristic, for example number of stock-items, it is convenient to name the variables using a symbolic representation of the common characteristic, and to distinguish the variables by an identification number included in the variable name. For example, if you are a newsagent you may wish to record your stock of six publications which may be typically the number of (a) *Your Computer* (b) *Reader's Digest* (c) *Popular Science* (d) *Trout and Salmon* (e) *Stamp Monthly* and (f) *She*. We require six variables for these items and, since the common characteristic is number of items, we can denote them as n(1), n(2), n(3), n(4), n(5) and n(6), as shown in Fig. 3.2.

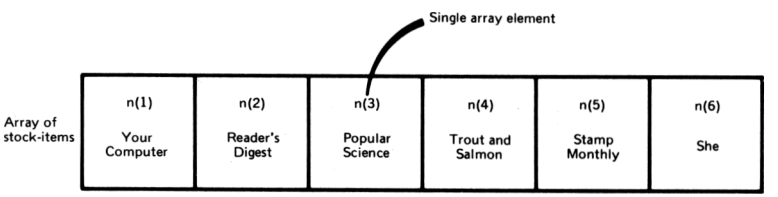


Fig. 3.2. One-dimensional array with six elements

The arrangement shown in Fig. 3.2 is known as a one-dimensional array containing six elements. To use this array in a Basic program it is necessary first to reserve the required memory locations for the array by using the **DIM** statement. In this example, as there are six elements in the one-dimensional array, the required **DIM** statement is **DIM** n(6). It is important to note that the number distinguishing the variables is enclosed in parentheses. This means that n(1), n(2), n(3), n(4), n(5) and n(6) are the uniquely named valid numeric variables. You may only use parentheses in a variable name when dealing with arrays.

We are not restricted to using arrays with one dimension. For example, it is possible to extend the one-dimensional array considered above to a two-dimensional form so that it can be used by

Newsagents	Your Computer	Reader's Digest	Popular Science	Trout and Salmon	Stamp Monthly	She
No.1	n(1,1)	n(1,2)	n(1,3)	n(1,4)	n(1,5)	n(1,6)
No.2	n(2,1)	n(2,2)	n(2,3)	n(2,4)	n(2,5)	n(2,6)
No.3	n(3,1)	n(3,2)	n(3,3)	n(3,4)	n(3,5)	n(3,6)
No.4	n(4,1)	n(4,2)	n(4,3)	n(4,4)	n(4,5)	n(4,6)

Fig. 3.3. Two-dimensional array with four rows and six columns

a wholesaler for magazine stock control. In Fig. 3.3 we show a two-dimensional array representing six stock-items required by four newsagents, and we see that this array contains 24 elements arranged in four rows and six columns. Each element is identified as a subscripted variable using the format

n(row number, column number)

For example, n(2, 4) refers to the number of *Trout and Salmon* ordered by newsagent No. 2. To use this array in a Basic program we must, of course, reserve the necessary memory locations at the start of the program by using

DIM n(4, 6)

The program listed below accepts data for the 24-element, two-dimensional array, allows you to change and examine the data, and outputs the total stock of each publication held by the wholesaler.

```
 5 DIM n(4,6)
 1Ø PRINT "You may select: " ' "1. Change array element " ' "2.
   Display array element " ' "3. Display total for any column"
 2Ø PRINT : PRINT "Input 1, 2 or 3"
 25 INPUT N
 35 IF N = 1 THEN GO TO 3ØØ
 45 IF N = 2 THEN GO TO 3ØØ
 55 IF N = 3 THEN GO TO 5ØØ
 65 CLS
 75 GO TO 1Ø
3ØØ PRINT "Input Row No. ";: INPUT R: PRINT R
3Ø2 IF R > 4 THEN GO TO 3ØØ
3Ø5 PRINT "Input Column No. ";: INPUT C: PRINT C
3Ø7 IF C > 6 THEN GO TO 3Ø5
31Ø IF N = 2 THEN GO TO 32Ø
312 PRINT "Input data for selected element": INPUT D: PRINT
   "Data is "; D
315 LET n(R,C) = D
317 PRINT : GO TO 1Ø
32Ø PRINT "Data in selected element is ";n(R,C)
325 PRINT : GO TO 1Ø
500 PRINT "You may select total for: " ' "1. Your Computer
   " ' "2. Reader's Digest " ' "3. Popular Science " ' "4. Trout
   and Salmon " ' "5. Stamp Monthly " ' "6. She"
525 PRINT "Input 1, 2, 3, 4, 5 or 6": INPUT T
53Ø LET Total = Ø
555 FOR R = 1 TO 4
56Ø LET Total = n(R,T) + Total
575 NEXT R
58Ø PRINT "Total for column ";T;" is = "; Total
59Ø GO TO 2Ø
```

We suggest that you load this program and investigate its operation.

Theoretically there is no restriction on the array size or number of array dimensions, but in practice you will find that the storage capacity of your ZX Spectrum memory is the limiting factor. Note that the number of memory locations that must be reserved for an array is the *product* of the total number of elements in each dimension. For example

DIM R(8, 10, 6)

requires 480 memory locations in order to store the data for this three-dimensional array.

Random numbers

You can use the **RANDOMIZE** keyword (abbreviated to **RAND** on the keyboard) with the **RND** function to generate numbers which have some degree of randomness, although they may not strictly satisfy a rigorous mathematical analysis for statistical randomness.

The **RND** function generates a defined sequence of 65 536 numbers in the range zero to 0.99998474, and the associated **RANDOMIZE** statement:

RANDOMIZE X

where X is a positive integer in the range 1 to 65 535, defines the starting point in the **RND** number sequence. However, for randomness it is desirable to have the **RND** function starting off at random points in its number sequence. This is achieved by omitting the number X, or by setting X = 0 or by omitting the **RANDOMIZE** statement. The **RANDOMIZE** statement is useful in program debugging when you are using the **RND** function in your program, and when you wish the program to follow the same generated sequence each time it is run. For example, using the statement

RANDOMIZE 45438

will start the defined sequence at the number zero. If you use the two-line program

```
1Ø RANDOMIZE 45438
2Ø PRINT RND : GO TO 2Ø
```

the first seven numbers of the sequence generated will be

```
Ø
Ø.ØØ112915Ø4
Ø.Ø8581543
Ø.43719482
Ø.79Ø25269
Ø.2691Ø83
Ø.18934631
```

If we omit the number after the **RANDOMIZE** statement then we obtain a different sequence each time we run the program, and the sequence appears random.

The program given below can be used to generate positive random numbers (well, at least they may be random!). This program requires you to input the maximum possible magnitude of a generated number. When you have done this, numbers of the sequence are generated and displayed.

```
15 DIM a$(12)
25 PRINT "Input desired maximum value"
35 INPUT n: PRINT "Maximum value = "; n: PRINT
65 PRINT "Press n-key for next number"
70 IF INKEY$ <> "n" THEN GO TO 70
72 LET x = INT (RND*n) + 1
74 PRINT AT 8,14;a$()
75 PRINT AT 8,0; "Next number = ";x
78 IF INKEY$ <> " " THEN GO TO 78
85 GO TO 70
```

If you input n = 6 you have a die, or if n = 55 this can be used for Football Pool Treble Chance selections (we hope you are lucky!).

Concluding remarks

By studying the material in this chapter you will have seen that the ZX Spectrum is a powerful number cruncher.

In the next chapter we will describe the string handling capability of the computer.

4

Strings

Introduction

When you wish to include text in your program you can use the **PRINT** statement followed by the text inserted in quotation marks. The text is a *string* of characters. For example, Good Morning! is a string of thirteen characters and this can be displayed on the screen using

PRINT "Good Morning!"

In a string you can use all the available characters on the keyboard except the quotation mark, because this will be interpreted as the end of the string. If you wish to include quotation marks within a string you may use single quotation marks (symbol shifted 7), and these will be printed within the string. For example

PRINT "Good Morning 'ZX Spectrum User'"

will display Good Morning 'ZX Spectrum User' starting in the top left-hand corner of the screen. If you require this string to be displayed elsewhere on the screen use the keyword **PRINT** followed by the **AT** pseudo-function in the format

PRINT AT row number, column number; "characters"

where rows are numbered from 0 to 21 starting from the top, and the columns are numbered from 0 to 31 starting from the left-hand side of the screen. For example

25 **PRINT AT** 21, 1; "Good Morning 'ZX Spectrum User' "

will print the string on the bottom line of the screen starting in column 1. Try changing the column number to 6 and see what happens.

When a string does not contain any characters between the quotation marks it is known as a null string.

Strings can be added together using the + operation. For example, try the following

PRINT AT 10, 4; "My name is " + "William"

Note that you may use only the + operation in this way.

String variables

You may assign a string to a variable, and to do this a single alphabetic character followed by a $ is used to denote the variable. However, note that a$ and A$ are indistinguishable, so you can only have a maximum of 26 string variables in your Basic program. The following simple program uses five string variables: A$, B$, C$, D$ and K$

```
15 LET A$ = "Happy Birthday"
25 LET B$ = " "
35 LET C$ = "DAD"
45 LET D$ = "MUM"
55 LET K$ = A$ + B$ + C$
65 PRINT AT 11, 8; K$
```

Exercise
Verify that the program displays Happy Birthday DAD. Change the string variable C$ to D$ in line 55 and re-run the program.

In the simple program above the string variables are defined using **LET** statements. However, it is possible to enter string variables using **INPUT** statements, which can take one of two forms

INPUT single alphabetic character $
or **INPUT LINE** single alphabetic character $

If you use an **INPUT** statement without **LINE** the computer operating system automatically displays double quotation marks on either side of the flashing L cursor, thereby prompting you to input the string. In contrast, when **LINE** is included in the **INPUT** statement the prompt to input the string is simply the flashing L cursor without quotation marks (as is the case for inputting a numeric variable).

Exercise
Change line 15 in the last program to 15 **INPUT** A$ and change line 35 to 35 **INPUT LINE** C$. Run the program and input the required strings.

Substrings

A substring is any set of consecutive characters extracted from an

existing string. To define a substring we use the string or string variable with the **TO** statement in the form

String
or
String variable } (Start **TO** Stop)

Therefore if the existing string is to be "ZX Spectrum User's Handbook", we can define this as the string variable A$

```
15 LET A$ = "ZX Spectrum User's Handbook"
```

We can then display the substring "Hand" by using

```
25 PRINT A$ (2Ø TO 23)
```

Alternatively, to print this substring in row 5 starting in column 13, we can use

```
25 PRINT AT 5, 13; A$ (2Ø TO 23)
```

To display the substring "m" we can use

```
25 PRINT A$ (11 TO 11)
```

If the Start position is omitted in defining the substring, it is assumed that we wish to start the substring at the first position in the string.

So to display "ZX Spectrum" we can use

```
25 PRINT A$ ( TO 11)
```

Similarly, if the Stop position of the substring is omitted it is assumed that the last position in the string defines the end of the substring. For example, to display "User's Handbook" we can use

```
25 PRINT A$ (13 TO )
```

When the substring Start position is greater than the Stop position a null substring is created.

Note that the Stop position of the substring must not exceed the position of the last character in the existing string. Furthermore you cannot use negative numbers to define the Start and Stop positions of a substring.

Exercise

If the string x$ = "London Bridge", verify that the substring "on" may be displayed using either

```
   PRINT x$ (2 TO 3)
or PRINT x$ (5 TO 6)
```

and show that the substring "ridge" may be displayed using

```
PRINT x$ (9 TO )
```

String functions

In Chapter 3 we saw that the ZX Spectrum can evaluate mathematical, trigonometrical and user defined mathematical functions. We shall now discuss the four string functions **LEN, VAL, VAL$** and **STR$**, and user defined string functions.

LEN

You can obtain a count of the LENgth of a string by using the **LEN** function. For example, the string "Tom and Jerry" has 13 characters and you can confirm this by entering

PRINT LEN "Tom and Jerry"

and you will see that the computer responds by displaying 13 on the screen. Similarly, if you input

PRINT LEN ("Tom" + "and" + "Jerry")

you will note that the computer responds by displaying 11 on the screen (we have not included the spaces between the words this time).

VAL

This function is used to eVALuate strings that are arithmetic expressions. For example

PRINT VAL "6↑6↑6"

evaluates $((6)^6)^6 = 6^{36} = 1.0314425E+28$.
You can also use the **VAL** function with graphic statements; for example

PRINT AT VAL "3↑2+2", **VAL**"4↑2";"©"

displays © in the centre of the screen.

Exercise
Show that **PRINT LEN** ("Today is" + "Monday") has the same value as **PRINT VAL** "3*5", that is 15.

VAL$

This function evaluates a string contained within three sets of double quotation marks and effectively the result is the same string

contained within a pair of double quotation marks. For example

VAL$ " " "Computers are fun" " " " is evaluated

as "Computers are fun" so that if we

PRINT LEN VAL$ " " "Computers are fun" " " "

it is the same as

PRINT LEN "Computers are fun"

Exercise
Confirm that 17 is the result of the two **PRINT LEN** statements above.

STR$

The **STR$** function allows you to convert decimal numbers into strings. For example, the statement

PRINT STR$ 562

displays 562 on the screen. It is equivalent to using the statement

PRINT "562"

Exercise
Confirm that the statement

PRINT LEN STR$ 691982.14

is the same as

PRINT LEN "691982.14"

User defined string functions

You may define your own string functions for use within a Basic program. These are *user defined string functions* and their general form is

line number **DEF FN** letter $ (argument $) = string

Three examples of valid forms of user defined string functions are

```
15 DEF FN a$() = "Mary"
25 DEF FN H$(b$) = "Mary" + b$
35 DEF FN T$(b$, c$) = "Mary" + b$ + c$
```

Note however that before we can use the user defined function in line 25 we must define the string variables b$, and before we can use line 35 we must also define the string variable c$.

Exercise

Enter and run the following program and investigate how the user defined functions are used.

```
  5 INPUT b$
 1Ø PRINT b$
 15 INPUT c$
 25 PRINT c$
115 DEF FN a$() = "Mary"
125 DEF FN H$(b$) = FN a$() + b$
135 DEF FN T$(b$, c$) = FN H$(b$) + c$
145 PRINT FN a$();";FN H$(b$)
16Ø PRINT FN T$(b$, c$)
```

String arrays

It is possible to set up string characters in an array. A one-dimensional array having N elements, where N is a positive integer, must be DIMensioned using a single alphabetic character followed by $ and the dimensions of the array. For example

DIM D$(1,N)

Each element is identified as a subscripted string variable using the format

D$(1,element number)

as shown in Fig. 4.1.

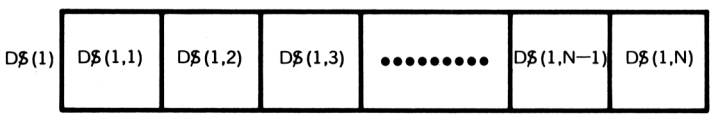


Fig. 4.1. One-dimensional string array

Consider the one-dimensional string "TELEVISION", which contains 10 characters. To define this string we can use the following

```
15 DIM D$(1,1Ø)
25 LET D$(1)="TELEVISION"
```

and this is assigned in the array as shown in Fig. 4.2. Therefore to print a single character from the string, say S, we use

PRINT D$(1,7)

Furthermore, to display on the screen the substring LEV we would use the statement

Fig. 4.2. One-dimensional string array for a string containing 10 characters

PRINT D$(1,3 **TO** 5)

while to print the entire string we can use the statement

PRINT D$(1)
or **PRINT** D$(1, **TO**)

Using this approach we can refer to individual characters within the string, to substrings, or to the entire string.

A two-dimensional array is constructed in a similar way. However, the DIMension statement must reserve sufficient memory for the longest string. For example if we have the four strings "January", "February", "March" and "April" we can dimension the array using

DIM M$ (4,8)

where 4 is the number of strings and 8 is the number of characters in the longest string. The arrangement of the corresponding two-dimensional array is shown in Fig. 4.3.

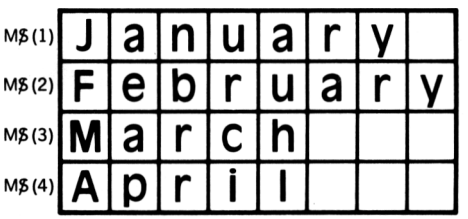


Fig. 4.3. Two-dimensional string array defining four strings

You can define this two-dimensional array using

```
1Ø DIM M$(4,8)
2Ø LET M$(1) = "January"
3Ø LET M$(2) = "February"
4Ø LET M$(3) = "March"
5Ø LET M$(4) = "April"
```

Exercise
Show that the substring "uary" may be obtained from the two-dimensional array above by using either

PRINT M$(1,4 **TO** 7)
or **PRINT** M$(1,4 **TO**)
or **PRINT** M$(2,5 **TO** 8)
or **PRINT** M$(2,5 **TO**)

It is worth noting that

PRINT M$(3,6 **TO**)

displays a null substring (three spaces) because M$(3,6), M$(3,7) and M$(3,8) are null.

It is also possible to have a string array with no dimensions! In such cases, strange as it may seem, the array must still be dimensioned using a single alphabetic character followed by $, and the number of null elements (spaces). For example,

DIM c$(5)

dimensions the null string array c$ which contains five spaces. This is an alternative and more efficient method (in terms of memory requirements) of defining spaces for use in text.

Exercise
Enter and run the following program to verify that the array c$(5) has the same effect as using the string S$.

```
 5 DIM c$(5)
10 LET a$ = "Name"
20 LET b$ = "Address"
30 LET S$ = "     "
40 PRINT a$ + S$ + b$
50 PRINT a$ + c$ + b$
```

Concluding remarks

By studying this chapter you should have learnt how to handle strings, string variables, substrings and string arrays. You should also have gained an understanding of the string functions **LEN, VAL, VAL$** and **STR$**, and seen how to define and use your own string functions.

You will find that the string handling capability of the ZX Spectrum provides you with a versatile and powerful method of displaying text.

5

Logical decisions

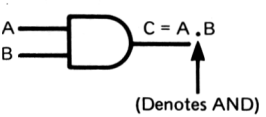
Introduction

In this chapter we show that logical operations and decisions can be implemented using logic gates and this may be useful to you when interfacing your ZX Spectrum to external hardware. We also show that logic operations may be performed on a bit-by-bit basis between two data bytes, which is applicable when writing machine-code programs. We conclude the chapter with a section on the conditional **IF** statement. This is a very useful feature of Basic, permiting complex decisions to be implemented in your programs.

AND, OR and NOT logic operations

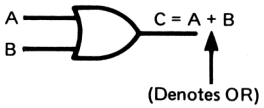
Your ZX Spectrum uses logic elements (electronic circuits) that give an output dependent on the value of one or more input variables. The input and output values are either logic level 0 (0V) or logic 1 (+5V).

The AND gate gives an output logic 1 when all the inputs are at logic 1, otherwise the output is logic 0. The symbol and truth table for a two-input AND gate are shown in Fig. 5.1.



Input		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 5.1. Symbol and truth table for a two-input AND gate

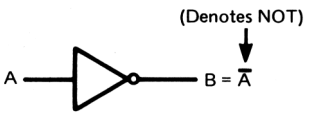


Input		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 5.2. Symbol and truth table for a two-input OR gate

The OR gate (Inclusive-OR) gives an output logic 1 when any, or any combinations, of the inputs is/are at logic 1. This implies that the output will only be at logic 0 when all the inputs are at logic 0. The symbol and truth table for a two-input OR gate are shown in Fig. 5.2.

The NOT gate (inverter or complement gate) gives an output which is the inverted (complemented) logic value of the input. The symbol and truth table are shown in Fig. 5.3.



Input	Output
A	B
0	1
1	0

Fig. 5.3. Symbol and truth table for the NOT gate

In the Z80A microprocessor the AND and OR operation are performed on a bit-by-bit basis between two data bytes. If the two data bytes are A = 10001010 and B = 11100111 the AND operation is achieved as

$$\begin{aligned} A &= 10001010 \\ B &= \underline{11100111} \\ \text{Result} = A.B &= \underline{10000010} \end{aligned}$$

and the OR operation yields

$$\begin{aligned} A &= 10001010 \\ B &= \underline{11100111} \\ \text{Result} = A + B &= \underline{11101111} \end{aligned}$$

The Z80A microprocessor can perform the NOT operation on a data byte using a CPL machine-code instruction, whereby it changes 0s to 1s and 1s to 0s. As an example, if A = 11011011 then the one's complement of A, ie NOT A, is

$$\begin{aligned} A &= 11011011 \\ \text{Result} = \overline{A} &= 00100100 = \text{NOT A} \end{aligned}$$

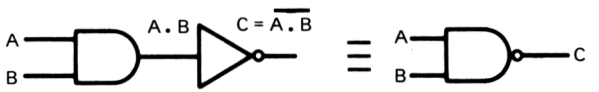
Exercise
Given two data bytes A = 10101011 and B = 11110111 show that on a bit-by-bit basis

$$A.B = 10100011$$
$$A + \overline{B} = 11111111$$
$$\overline{A} = 01010100$$
$$A + \overline{B} = 10101011 = A$$

Note that you can use **AND, OR** and **NOT** in conditional **IF** statements in your Basic programs. However, in such cases you should realise that, instead of interpreting the logic operations in terms of 0s and 1s, the ZX Spectrum interprets them as false and true respectively. This is discussed more fully later.

NAND and NOR logic operations

You can implement these logic operations using gate circuits. The NAND gate is obtained by combining an AND gate with a NOT gate. This is shown in Fig. 5.4 for the case of a two-input NAND element. This gate only gives an output logic 0 when all the inputs are at logic 1; for all other input combinations it gives an output logic 1. The symbol and truth table for a two-input NAND gate are given in Fig. 5.4.

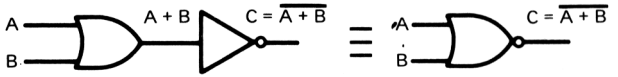


Input		Output
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 5.4. Symbol and truth table for a two-input NAND gate

The NOR gate is obtained by combining an OR gate with a NOT gate. This is shown in Fig. 5.5 for the case of a two-input NOR element. This gate only gives an output logic 1 when all the inputs are at logic 0; for all other input combinations it gives an output logic 0. The symbol and truth table for a two-input NOR gate are given in Fig. 5.5.

Let us consider how these logical operations work for two data



Input		Output
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 5.5. Symbol and truth table for a two-input NOR gate

bytes on a bit-by-bit basis. Using A = 11110101 and B = 01111100 we obtain

$$\begin{array}{rr} A = & 11110101 \\ B = & \underline{01111100} \\ \text{Result} = \overline{A.B} = & \underline{10001011} \end{array} \qquad \begin{array}{rr} A = & 11110101 \\ B = & \underline{01111100} \\ \text{Result} = \overline{A + B} = & \underline{00000010} \end{array}$$

It is also interesting to note that the NAND and NOR operations on the numeric variables A and B may be replaced by an equivalent form, obtained using DeMorgan's theorems, namely

(NAND) $\overline{A.B} = \overline{A} + \overline{B}$
(NOR) $\overline{A + B} = \overline{A}.\overline{B}$

and these are shown in logic diagram form in Fig. 5.6.

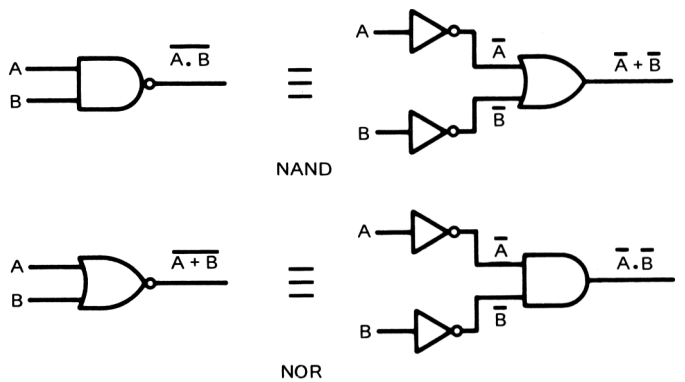


Fig. 5.6. Gate equivalents obtained using DeMorgan's theorems

Exercises

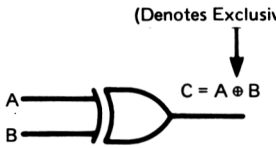
1. Given two data bytes A = 10111000 and B = 00111001 show that on a bit-by-bit basis

$\overline{A.B}$ = 11000111
$\overline{A + B}$ = 01000110

2. By using DeMorgan's theorems prove that
$A + B = \overline{\overline{A}\,\overline{B}}$
$A.B = \overline{\overline{A}+\overline{B}}$

Exclusive-OR logic operation

You can implement this operation using a gate circuit. The Exclusive-OR gate gives an output logic 1 when any, or any combination, of the inputs, but *excluding* the combination of all inputs, is/are at logic 1. This implies that the output will only be logic 0 when all the inputs are at the same logic level, either logic 1 or logic 0. The symbol and truth table for a two-input Exclusive-OR gate are shown in Fig. 5.7.



Input		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 5.7. Symbol and truth table for a two-input Exclusive-OR gate

This logic operation can be performed for two data bytes on a bit-by-bit basis. If we consider A = 10000111 and B = 01110000 we obtain

A = 10000111
B = 01110000
Result = A ⊕ B = 11110111

The Exclusive-OR operation can be performed as a machine-code instruction in the Z80A using an XOR instruction.

Exercise
Given two data bytes, A = 11100011 and B = 10101010, show that on a bit-by-bit basis
$A \oplus B$ = 01001001

Conditional IF statement

In a Basic program you may find that you require decisions to be made regarding the existence of desired conditions. To do this you

can use conditional **IF** statements, which must be written as

IF condition **THEN** statement

In the Basic instruction the statement following **THEN** can be used conditionally to direct the program to another line, for example by means of a **GO TO** statement, otherwise if the stated condition is not met, then the program proceeds to the next line in the program. This can be appreciated by considering the statements

```
2Ø IF h >= 12 AND h <= 112 THEN GO TO 45
3Ø STOP
```

where this will be interpreted to mean if the numeric variable, h, is in the range 12 to 112 inclusive then go to line 45 of the program, otherwise go to the next line of the program, ie to **STOP**. We can represent this in the form of the following truth table.

h >= 12 (h greater than or equal to 12?)	h <= 112 (h less than or equal to 112?)	Program control goes to
True	True	Line 45
True	False	Line 30
False	True	Line 30
False	False	Line 30

The condition of mathematical expressions or numbers can be established using the following mathematical relationships. Also shown are the ZX Spectrum symbols representing them.

equal to:	symbol =
not equal to:	symbol <>
less than:	symbol <
less than or equal to:	symbol <=
greater than:	symbol >
greater than or equal to:	symbol >=

The program listed below shows a method of investigating the effects of using these relationships.

```
 5 PRINT
15 PRINT "Input Values for A and B"
25 INPUT A
35 INPUT B
45 CLS
55 IF condition THEN GO TO 85
65 PRINT "Condition False"
```

```
75 GO TO 5
85 PRINT "Condition True"
95 GO TO 5
```

We suggest you enter the program and for your first attempt insert between **IF** and **THEN** the condition A = B in line 55 of the program. On running the program you will note that for entered values of A equal to B the program goes to line 85, while for values of A not equal to B the program goes to line 65.

Exercise
Change line 55 in the above program to test each of the other conditional relationships

```
A <> B
A < B
A <= B
A > B
A >= B
```

It is also possible to use the conditional **IF** statement to determine the condition of strings. The conditional relationships considered above can be used to establish whether two strings are identical (equal) or whether the alphabetical order of one string precedes that of another string. Assuming that the two strings involved with the **IF** statement are A$ and B$, we can summarise the possible conditional tests as follows:

```
A$ = B$:    A$ is identical to B$
A$ <> B$:   A$ is not identical to B$
A$ > B$:    A$ follows B$ in alphabetical order
A$ < B$:    A$ precedes B$ in alphabetical order
A$ >= B$:   A$ follows B$ in alphabetical order
              or A$ is identical to B$
A$ <= B$:   A$ precedes B$ in alphabetical order
              or A$ is identical to B$
```

For example, if we use

```
55 IF A$<B$ THEN GO TO 85
65 next line of program
```

this is interpreted by the ZX Spectrum to mean: if the string A$ precedes the string B$ in alphabetical order then go to program line number 85, otherwise go to the next line (line number 65) of the program.

The program listed below, in which you insert the required condition in line 55, provides you with a method of investigating the above conditional tests on the two strings.

```
 5 PRINT
15 PRINT "Input strings A$ and B$"
25 INPUT A$
35 INPUT B$
45 CLS
55 IF condition THEN GO TO 85
65 PRINT "Condition False"
75 GO TO 5
85 PRINT "Condition True"
95 GO TO 5
```

It is possible to combine decision-making relationships by using logical operations. The ZX Spectrum Basic has the logical operations **AND, OR** and **NOT** available for this purpose.

The **AND** operation is used to test that *all* relationships used in the conditional **IF** statement are true. For example, if we change line 55 in the above program to

```
55 IF A$ = "Z" AND B$ = "W" THEN GO TO 85
```

the program goes to line 85 only when the string A$ is equal to the single character Z *and* the string B$ is equal to the single character W, otherwise the next line of the program is used.

The **OR** operation is used to test that *at least one* relationship used in the conditional **IF** statement is true. For example, if we change line 55 in the previous program to

```
55 IF A$ = "Z" OR B$ = "W" THEN GO TO 85
```

the program goes to line 85 when *either* the string A$ is equal to the single character Z *or* the string B$ is equal to the single character W, *or* both conditions are true, otherwise the next line of the program is used.

The **NOT** function is used to test that the relationship following it is not true. For example, if we change line 55 in the previous program to

```
55 IF NOT A$ = "Z" AND B$ = "W" THEN GO TO 85
```

the program goes to line 85 when the string A$ is *not* equal to the single character Z *and* the string B$ is equal to the single character W, otherwise the next line of the program is used.

Concluding remarks

In this chapter you have learned how logic operations and decisions can be implemented using logic gates (hardware) and conditional **IF** statements (software). It is important to have the logical process(es) defined unambiguously and in this respect it is often appropriate to draw a flowchart. This is covered in the next chapter.

6

Flowcharts, loops and subroutines

Introduction

The first consideration in preparing a relatively complex program is to establish the logical sequence of steps to be performed by the microcomputer to achieve the desired result. The initial preparation may involve drawing a *flowchart* which illustrates diagrammatically the program steps, and includes the loops and subroutines involved. Each step in the logical sequence of the flowchart has then to be converted to appropriate Basic functions or statements.

Flowcharts

A flowchart is made up of graphical symbols that are connected together by straight lines. Arrow-heads are drawn on the connecting lines to indicate the *direction of flow* in the program. A selection of commonly used flowchart symbols is shown in Fig. 6.1.

The start and end of the flowchart is indicated by the *terminal* symbol shown in Fig. 6.1(a) and obviously it is connected to the flowchart by only one line. The *rhomboid* symbol shown in Fig. 6.1(b) is used to show a specific operation by an input or output device. The *rectangle* symbol shown in Fig. 6.1(c) is used to indicate that a specific action is to be taken. A statement inside the rectangle specifies the action, and this may be stated in plain English or, alternatively, it could be a logical or algebraic expression. The *diamond* symbol shown in Fig. 6.1(d) is used to indicate the point in a program at which a decision has to be made. For example, it may be necessary at some point in the program to know whether or not the numeric variable A is greater than zero (see Fig. 6.1(d)) and obviously the answer to this question is either no or yes. Clearly the program

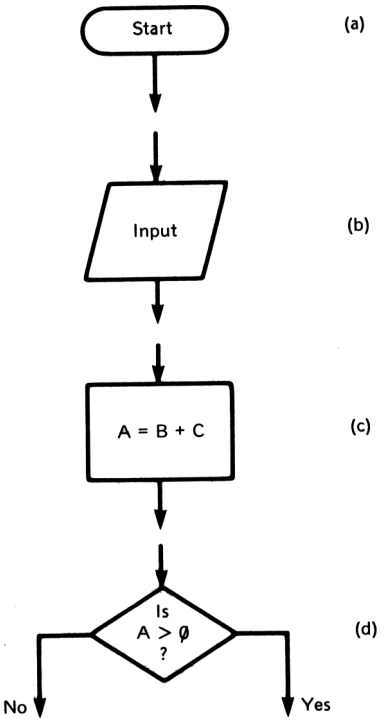


Fig. 6.1. Flowchart symbols (a) terminal symbol (b) input/output symbol (c) action symbol (d) decision symbol

will proceed along one of two possible paths depending on the decision made.

As an example, Fig. 6.2 shows the basic flowchart that can be used to represent the generation of a time delay. This flowchart contains two loops, so it is now the right time to consider loop operations before carrying on with the translation of the flowchart into a program.

Loops

A loop is a sequence of instructions that the computer repeats. It is possible to make it repeat the sequence a specified number of times using the **FOR, NEXT, TO** and **STEP** commands in the format

FOR variable = initial value **TO** final value **STEP** step size
(sequence of instructions)
NEXT variable

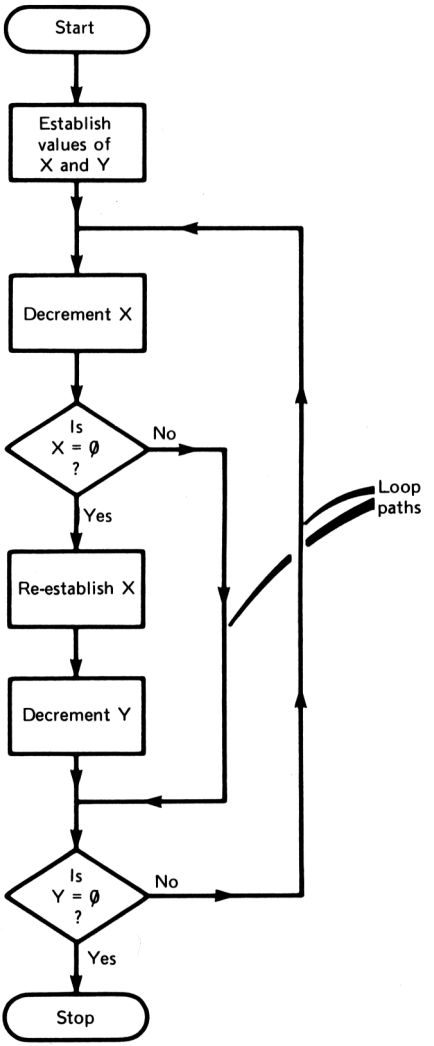


Fig. 6.2. Flowchart for a time delay program

For example, if we have a one-dimensional numeric array containing six elements which we wish to initialise with the appropriate data, then we can use the following loop to input the data

```
1Ø DIM C(6)
2Ø FOR X = 1 TO 6
3Ø INPUT d
```

```
35 PRINT d
4Ø LET C(X) = d
5Ø NEXT
```

You will note that line 2Ø does not include **STEP** and step size so, by default, the computer takes the step size to be 1.

The program works by initially letting X = 1 and then implements lines 3Ø to 4Ø inclusive, that is, inputs the data for element C(1) of the array; line 5Ø directs the program back to line 3Ø (looping operation) and takes X to be the initial value incremented by the step size, thereby making the computer go through the sequence again to input data for array element C(2). The program continues looping until C(6) has been entered.

Exercise
Change line 2Ø in the above program to

```
2Ø FOR X = 6 TO 1 STEP – 1
```

and verify that the program initialises the one-dimensional array in reverse order.

An alternative approach to implementing a looping operation is to use a **GO TO** statement with the conditional **IF** statement discussed in Chapter 5.

To implement the operation for initialising our six-element array, we can achieve the looping operation by using the **GO TO** and **IF** statements in this form

```
1Ø DIM C(6)
2Ø LET X = 1
3Ø INPUT d
35 PRINT d
4Ø LET C(X) = d
5Ø LET X = X + 1
6Ø IF X < > 7 THEN GO TO 3Ø
```

Exercise
Change lines 2Ø, 5Ø and 6Ø in the program to

```
2Ø LET X = 6
5Ø LET X = X – 1
6Ø IF X < > Ø THEN GO TO 3Ø
```

and verify that the program initialises the one-dimensional array in reverse order.

Translating the flowchart into a program

A flowchart that can be used to represent the generation of a time delay is given in Fig. 6.2. Note that the delay duration is dependent upon the values assigned to the positive integer numeric variables X and Y. We suggest you study this flowchart to understand the significance of each step in the sequence of operations. You will find it convenient to assign values to X and Y, say 3 and 2 respectively, and to work out with pencil and paper the intermediate values of X and Y after each step. Note that by changing the values of X and Y, the number of steps is correspondingly changed, hence the time required to execute the sequence of operations can be varied, and consequently from Start to Stop there is a variable time delay.

For each step in the logical sequence of the flowchart we have to select appropriate Basic functions or statements which can be used to implement the desired operation.

To illustrate the principles involved let us write a time delay program corresponding to the flowchart shown in Fig. 6.2. The steps involved in translating the program are tabulated below.

Flowchart statement	*Appropriate Basic instructions*
Start	Manual initiation of program by **RUN** and **ENTER**
Establish values of X and Y	**INPUT** X **INPUT** Y
Decrement X	**LET** X = X − 1
Is X = Ø?	**IF** X < > Ø **THEN GO TO** line number
Re-establish X	**LET** X = S (S will have to be set to the initial value of X)
Decrement Y	**LET** Y = Y − 1
Is Y = Ø?	**IF** Y < > Ø **THEN GO TO** line number
Stop	**STOP**

To write the program each instruction obviously requires a line number (it is wise to leave plenty of space between line numbers to allow additional lines to be inserted if necessary), and the correct line number must be inserted in the conditional **IF** statement program lines. Also you will have noted that the initial value of X must be saved, because it must be re-established during program execution, and this is achieved by letting S = X immediately after X is entered.

The program is listed below; appropriate **PRINT** statements have been included to display suitable captions to assist the user.

```
  5 PRINT "Time Delay Demonstration Program " ' ' " Input
    Positive Integer X"
 15 INPUT X
 25 PRINT X
 55 LET S = X
 65 PRINT "Input Positive Integer Y"
 75 INPUT Y
 85 PRINT Y
 95 LET X = X - 1
115 IF X < > Ø THEN GO TO 135
12Ø LET X = S
125 LET Y = Y - 1
135 IF Y < > Ø THEN GO TO 95
145 PRINT "End of delay"
```

We suggest you run this program with X = Y = 25. You will find that the time required to execute the program is approximately 10 seconds. Try other values of X and Y but do not be tempted to make the numbers too large unless you have plenty of time to wait for the "End of Delay" response.

The above program must have positive integer values for X and Y, and to ensure that non-integer, non-zero and negative numbers are rejected by the program you can insert the following two additional lines

```
2Ø IF X < = Ø OR (X - INT X) < > Ø THEN GO TO 5
8Ø IF Y < = Ø OR (Y - INT Y) < > Ø THEN GO TO 65
```

We have used this form of time delay program in Program 2 in Appendix A to generate an approximate one-second delay, which is incorporated in a pseudo 24-hour stop-watch. You may find it interesting to try this program for yourself.

Subroutines

In many programs you will find that the same set of instructions may be required more than once, and in such cases it is sensible to implement this common set of instructions as a subroutine.

A subroutine is a set of program instructions that can be reached (*called*) from more than one place in the main program. It is normally placed at the beginning or end of the main program. The process is illustrated in Fig. 6.3. To explain the principle we will consider a timer problem which uses a time-delay subroutine in implementing a solution.

The problem under consideration is to use the ZX Spectrum to

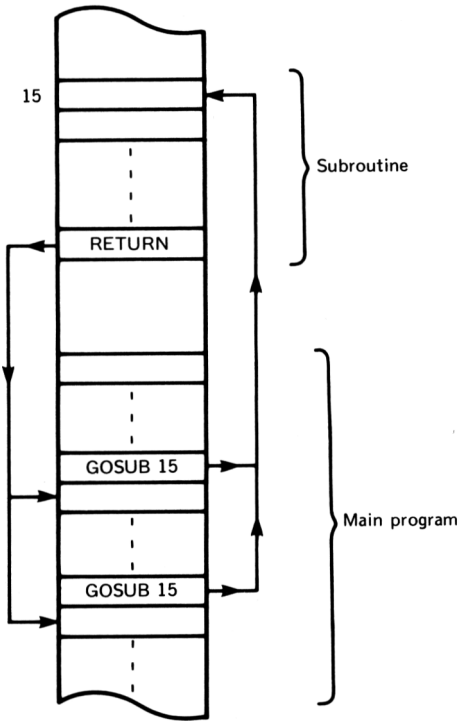


Fig. 6.3. Illustration of subroutine GOSUB and RETURN

display a positive integer count-value which decrements every second until it becomes zero. The displayed count-value must indicate the number of seconds remaining in each equal half-period, and indicate which half-period is being timed out.

In the program the subroutine (lines 15Ø-265) is used to create the required time delay of approximately one second. You can see that the corresponding common set of instructions are called by using the **GOSUB** statement at two points in the main program, lines 55 and 115, and the necessary **RETURN** (see Fig. 6.3) is placed at the end of the subroutine, at line 265.

You may find the following program useful as a timer for competitive games or sports which may have predetermined time periods, for example chess, ice-hockey, Scrabble, etc.

```
 5 PRINT "Input number of seconds in each half"
1Ø INPUT N
12 IF N-INT N > < Ø THEN GO TO 135
2Ø CLS
```

```
 24 PRINT "Number of seconds remaining" ' "in First half is "
 25 FOR J = N TO Ø STEP -1
 55 GO SUB 15Ø
 65 NEXT J
 66 CLS
 7Ø PRINT "Number of seconds remaining" ' "in Second half
    is "
 75 FOR J = N TO Ø STEP -1
115 GO SUB 15Ø
12Ø NEXT J
125 STOP
135 CLS
145 GO TO 5
15Ø PRINT AT 1, 18;"        "
17Ø PRINT AT 1, 18; J
175 LET X = 14
185 LET S = X
195 LET Y = 4
215 LET X = X - 1
225 IF X < > Ø THEN GO TO 255
235 LET X = S
245 LET Y = Y - 1
255 IF Y < > Ø THEN GO TO 215
265 RETURN
```

Concluding remarks

In this chapter you have learnt that flowcharts assist you in determining the required program steps and that the sequence of steps summarised in the flowchart can be readily translated into a Basic program.

You will find that the efficiency of your programs can often be improved by using appropriate loops and subroutines.

7

Colour graphics

Introduction

You can create colour pictures on a colour television screen using the ZX Spectrum's eight available colours (see Plate 7.1). The actual colours are shown above the number keys 0 to 7 and are listed below.

Key	*Colour*	
0	black (maximum grey scale level)	
1	blue	
2	red	
3	magenta	decreasing
4	green	shades of
5	cyan	grey
6	yellow	
7	white (minimum grey scale level)	

If you are using a monochrome television then the number keys 0 to 7 can be used to select and display appropriate shades of grey - this is indicated in the above list.

The region of the screen where you display characters is called the paper and the surrounding region is referred to as the border. For example, if you use

BORDER 3 : **PAPER** 6

and **ENTER** these commands, then *only* the border colour, magenta, is displayed; the paper colour is unaltered. However, on pressing **ENTER** again the paper now changes to yellow. This occurs because the **PAPER** command changes the *attribute* for the paper but the colour of the paper will only change when the computer is

subsequently commanded to output characters to the screen.

You can investigate this by entering and running the following program.

```
 5 BORDER 3
10 PAPER 6
15 CLS
18 STOP
20 PAPER 1
35 PRINT "***"
40 STOP
45 PAPER 4
80 LIST
```

In this program you may note that line 5 sets and displays a magenta border, and line 10 initialises the paper attribute for all characters to 6 (yellow). Line 15 commands the computer to output character spaces to the paper using their current attributes, hence the entire paper changes to yellow. To continue you enter **CONT**inue and you will now note that line 20 sets the paper attribute for all characters to 1 (blue), and line 35 is used to print the characters *** which now appear on blue paper as defined by their current attributes. Note that, since line 35 only requests three characters to be displayed, then only their paper background colour is changed.

When you continue by entering **CONT**inue you will see that in line 45 the paper attribute is set to 4 (green) and line 80 commands the computer to list the program. Only the screen characters for the listing are displayed on green paper because these are the only characters put on the paper by the **LIST** command.

You will note that the characters displayed by the above program are all black, ie the ink is black. To change the colour of the characters you use the **INK** command followed by the required colour number. For example if you enter

```
INK 2
```

all subsequently displayed characters will be in red ink until the **INK** attribute is redefined.

In addition to using the numbers 0 to 7 to define a colour you may use the numbers 8 and 9 after the **PAPER** and **INK** commands. The number 8 is used with these commands to mean 'no change'; that is, the colour of the paper or ink is left as it was earlier. The number 9 is used with these commands to mean 'contrast'; that is, the paper or ink is made white if the other is black, blue, red or magenta, or it is made black if the other is green, cyan, yellow or white.

You may wish to have displayed characters flashing or steady and this is set by the **FLASH** command followed by 0, 1 or 8, where 0

defines a steady display, 1 defines a flashing display, and 8 defines a no-change mode. When characters are flashing the ink and paper colours are interchanged periodically.

The **BRIGHT** command can be used to set displayed characters with normal or enhanced intensity. This is set up by the **BRIGHT** command followed by Ø, 1 or 8, where Ø defines a normal display, 1 defines an enhanced intensity display, and 8 defines a no-change mode.

When you wish to exchange paper and ink colours (inverse video) you can use the command

INVERSE 1

and to revert back to the normal paper and ink (normal video) colours you may use the command

INVERSE Ø

Another useful command is the **OVER** command which when entered as

OVER 1

permits new characters to be printed on top of existing characters. For further details see the next section. The command

OVER Ø

obliterates existing characters when new characters are printed.

In the next section we examine the character set of the ZX Spectrum and describe how you can define your own characters.

The character set

The character set of the ZX Spectrum is summarised in Table 7.1. You will note that each character is assigned a unique decimal code in the range Ø to 255, but some codes are not used and others are used as control characters. There are 224 characters that can be displayed on the screen, ie codes 32 to 255 inclusive.

In applications where we require the given code of the first character in a string, we use the **CODE** function. Conversely, when we want to obtain a character from its code we use the **CHR$** function. If you execute

PRINT CODE "Rocket"

you will see that the code 82, corresponding to R, is displayed on the screen. If you execute

PRINT CHR$ 36

you will see the $ character displayed on the screen.

Table 7.1. ZX Spectrum character set summary

Code	*Character*	*Hex*	*Code*	*Character*	*Hex*
0 to 5	Not used		63	?	3F
6	**PRINT** comma	06	64	@	40
7	**EDIT**	07	65	A	41
8	cursor left	08	66	B	42
9	cursor right	09	67	C	43
10	cursor down	0A	68	D	44
11	cursor up	0B	69	E	45
12	**DELETE**	0C	70	F	46
13	**ENTER**	0D	71	G	47
14	number	0E	72	H	48
15	not used	0F	73	I	49
16	**INK** control	10	74	J	4A
17	**PAPER** control	11	75	K	4B
18	**FLASH** control	12	76	L	4C
19	**BRIGHT** control	13	77	M	4D
20	**INVERSE** control	14	78	N	4E
21	**OVER** control	15	79	O	4F
22	**AT** control	16	80	P	50
23	**TAB** control	17	81	Q	51
24 to			82	R	52
31	Not used		83	S	53
32	space	20	84	T	54
33	!	21	85	U	55
34	"	22	86	V	56
35	#	23	87	W	57
36	$	24	88	X	58
37	%	25	89	Y	59
38	&	26	90	Z	5A
39	'	27	91	[	5B
40	(	28	92	/	5C
41	)	29	93	]	5D
42	*	2A	94	↑	5E
43	+	2B	95	_	5F
44	,	2C	96	£	60
45	–	2D	97	a	61
46	.	2E	98	b	62
47	/	2F	99	c	63
48	0	30	100	d	64
49	1	31	101	e	65
50	2	32	102	f	66
51	3	33	103	g	67
52	4	34	104	h	68
53	5	35	105	i	69
54	6	36	106	j	6A
55	7	37	107	k	6B
56	8	38	108	l	6C
57	9	39	109	m	6D
58	:	3A	110	n	6E
59	;	3B	111	o	6F
60	<	3C	112	p	70
61	=	3D	113	q	71
62	>	3E	114	r	72

Code	Character		Hex
115	s		73
116	t		74
117	u		75
118	v		76
119	w		77
120	x		78
121	y		79
122	z		7A
123	{		7B
124	|		7C
125	}		7D
126	~		7E
127	©		7F
128	□		80
129	▗		81
130	▖		82
131	▄		83
132	▟		84
133	▌		85
134	▚		86
135	▛		87
136	▘		88
137	▞		89
138	▐		8A
139	▜		8B
140	▝		8C
141	▟		8D
142	▀		8E
143	■		8F
144	(a)		90
145	(b)		91
146	(c)		92
147	(d)		93
148	(e)		94
149	(f)		95
150	(g)		96
151	(h)		97
152	(i)		98
153	(j)		99
154	(k)	user	9A
155	(l)	graphics	9B
156	(m)		9C
157	(n)		9D
158	(o)		9E
159	(p)		9F
160	(q)		A0
161	(r)		A1
162	(s)		A2
163	(t)		A3
164	(u)		A4
165	**RND**		A5
166	**INKEY$**		A6
167	**PI**		A7
168	**FN**		A8

Code	Character	Hex
169	**POINT**	A9
170	**SCREEN$**	AA
171	**ATTR**	AB
172	**AT**	AC
173	**TAB**	AD
174	**VAL$**	AE
175	**CODE**	AF
176	**VAL**	B0
177	**LEN**	B1
178	**SIN**	B2
179	**COS**	B3
180	**TAN**	B4
181	**ASN**	B5
182	**ACS**	B6
183	**ATN**	B7
184	**LN**	B8
185	**EXP**	B9
186	**INT**	BA
187	**SQR**	BB
188	**SGN**	BC
189	**ABS**	BD
190	**PEEK**	BE
191	**IN**	BF
192	**USR**	C0
193	**STR$**	C1
194	**CHR$**	C2
195	**NOT**	C3
196	**BIN**	C4
197	**OR**	C5
198	**AND**	C6
199	**<=**	C7
200	**>=**	C8
201	**<>**	C9
202	**LINE**	CA
203	**THEN**	CB
204	**TO**	CC
205	**STEP**	CD
206	**DEF FN**	CE
207	**CAT**	CF
208	**FORMAT**	D0
209	**MOVE**	D1
210	**ERASE**	D2
211	**OPEN #**	D3
212	**CLOSE #**	D4
213	**MERGE**	D5
214	**VERIFY**	D6
215	**BEEP**	D7
216	**CIRCLE**	D8
217	**INK**	D9
218	**PAPER**	DA
219	**FLASH**	DB
220	**BRIGHT**	DC
221	**INVERSE**	DD
222	**OVER**	DE

223	**OUT**	DF	240	**LIST**	F0
224	**LPRINT**	E0	241	**LET**	F1
225	**LLIST**	E1	242	**PAUSE**	F2
226	**STOP**	E2	243	**NEXT**	F3
227	**READ**	E3	244	**POKE**	F4
228	**DATA**	E4	245	**PRINT**	F5
229	**RESTORE**	E5	246	**PLOT**	F6
230	**NEW**	E6	247	**RUN**	F7
231	**BORDER**	E7	248	**SAVE**	F8
232	**CONTINUE**	E8	249	**RANDOMIZE**	F9
233	**DIM**	E9	250	**IF**	FA
234	**REM**	EA	251	**CLS**	FB
235	**FOR**	EB	252	**DRAW**	FC
236	**GO TO**	EC	253	**CLEAR**	FD
237	**GO SUB**	ED	254	**RETURN**	FE
238	**INPUT**	EE	255	**COPY**	FF
239	**LOAD**	EF			

Exercise
Use the program below to obtain a display of all the characters that can be displayed on the screen (see plate 7.2).

```
10 FOR x = 255 TO 32 STEP -1
15 PRINT CHR$ x;
25 NEXT x
```

To enter the 16 graphics symbols, codes 128 to 143 in Table 7.1, the computer must be in the graphics mode (Cursor **G**). To establish the graphics mode press **GRAPHICS** (**CAPS SHIFT** and 9), and to exit from graphics mode press **GRAPHICS** again. Note that to obtain the eight graphic symbols, codes 128 to 135, you press the appropriate key, 1 to 8, but to obtain their inverse graphic symbols, codes 136 to 143, you press the appropriate key, 1 to 8, with the **SYMBOL SHIFT** key held down.

Exercise
Enter

PRINT CODE "■"

and verify that 143 is displayed.

Characters stored in read only memory (ROM)

The characters listed in Table 7.1 that have codes in the range 32 to 127 inclusive are stored in a block of read only memory (ROM) between memory addresses 15616 to 16383 inclusive. Each character is defined by the contents of eight successive memory bytes. The space character (code 32) is stored in the eight bytes in the

address range 15616 to 15623, and successive codes are correspondingly stored in successive blocks of eight bytes.

The eight bytes of any of these characters are stored in memory from address 15616 + (8*(code–32)) to address 15623 + (8*(code–32)). For example, the + character (code 43) is stored in the memory address range 157Ø4 to 15711. Let us peek into these memory locations using

```
1Ø FOR x = Ø TO 7
15 PRINT PEEK (157Ø4 + x)
2Ø NEXT x
25 STOP
```

to obtain a display of the contents of each memory location. The values are

Memory address	*Displayed contents (Decimal)*
157Ø4	Ø
157Ø5	Ø
157Ø6	8
157Ø7	8
157Ø8	62
157Ø9	8
1571Ø	8
15711	Ø

You may recall from Chapter 2 that the contents of a memory location are stored as an eight-bit binary word, so the + character is stored in eight bytes of memory in binary form as

Memory address	*Memory contents (Binary)*
157Ø4	00000000
157Ø5	00000000
157Ø6	00001000
157Ø7	00001000
157Ø8	00111110
157Ø9	00001000
1571Ø	00001000
15711	00000000

When the computer displays this + character it interprets the bit pattern such that 0 is paper and 1 is ink, and it is displayed using an 8 × 8 picture-element (pixel) grid, as shown in Fig. 7.1.

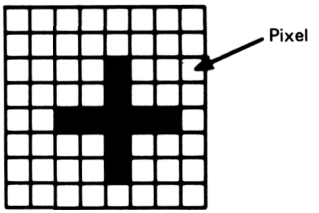


Fig. 7.1. Pixel grid definition of the + character

Exercise
Verify that the £ character (code 96) is displayed in the pixel grid form shown in Fig. 7.2.

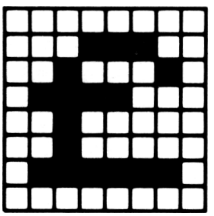


Fig. 7.2. Pixel grid definition of the £ character

You can use Program 3 in Appendix A to display, for an entered character code, the 8 × 8 bit character definition and the pixel grid definition magnified by a factor of 64. Plate 7.3 shows an example of the output form produced by this program.

User defined characters

In the previous section we saw how a set of characters was defined and stored in read only memory (ROM). These characters *cannot* be changed but when you require additional characters you may define them and store them in the area of volatile random access memory (RAM) provided.

You may define up to 21 *user defined characters*, which requires 168 bytes of memory, and the area of RAM reserved for them has the address range 32600 to 32767 in the 16K Spectrum and the address range 65368 to 65535 in the 48K Spectrum.

You can obtain the address of the first byte in memory of any of the user defined graphic characters by using the **USR** function in the form

PRINT USR "user defined graphic symbol"

and the computer outputs to the screen the appropriate address. For example

PRINT USR "D"

outputs the address 32624 for the 16K Spectrum, and 65392 for the 48K Spectrum.

The area of RAM used for the user defined characters is initialised, at switch-on, with data bytes which define the alphabetic characters A to U. The eight bytes of any of these characters are stored in the memory from address Addr + 8*(code–144) to address Addr + 7 + 8*(code–144), where code has a value in the range 144 to 164, corresponding to user defined characters A to U respectively, as given in Table 7.1, and Addr is equal to 32600 for the 16K Spectrum and 65368 for the 48K Spectrum.

For example, in the 16K Spectrum, if you examine the content of the eight bytes of memory with addresses from 32672 to 32679 inclusive, you will see that the J character is stored in binary form as

Memory address	*Memory contents (Binary)*
32672	00000000
32673	00000010
32674	00000010
32675	00000010
32676	01000010
32677	01000010
32678	00111100
32679	00000000

A simple way of proving this to yourself is to run Program 3, given in Appendix A, with line 30 changed to

30 **LET** j = 32599 + 8*(code–144) + x (16K Spectrum)
or 30 **LET** j = 65367 + 8*(code–144) + x (48K Spectrum)

Exercise
Run Program 3 given in Appendix A, with line 30 changed to the form above, and verify that the initialised user defined graphics characters for code values 145 and 162 are B and S respectively.

The initialised user defined characters will probably be of very little use to you because the same characters, A to U, exist as part of the character set stored in ROM. However, because the user defined characters are stored in RAM you can generate your own character by writing the appropriate eight bytes into any of the 21 user defined graphics memory areas. For example, to obtain a character of the

form shown in Fig. 7.3 you must first define the bit pattern of each byte of the character. In this example we have

00001000
00010100
00100010
01000001
00100010
00010100
00001000
00000000

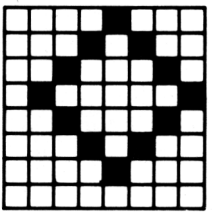


Fig. 7.3. User defined graphics character for diamond

Let us assume that this character is to be identified as the user defined character D. We must store the above bytes in memory addresses Addr to Addr + 7, where Addr is equal to 32624 for the 16K Spectrum or 65392 for the 48K Spectrum. We can do this using

POKE Addr, **BIN** ØØØØ1ØØØ
POKE Addr + 1, **BIN** ØØØ1Ø1ØØ
⋮ ⋮
POKE Addr + 7, **BIN** ØØØØØØØØ

To examine the binary bit pattern and pixel grid definition of this defined character we can again use Program 3 in Appendix A, with line 3Ø changed to

3Ø **LET** j = 32599 + 8*(code–144) + x (16K Spectrum)
or 3Ø **LET** j = 65367 + 8*(code–144) + x (48K Spectrum)

To include a user defined graphic character in a Basic program you enter graphics mode and access the character by pressing the appropriate alphabetic symbol key, A to U. For the above example, to display the diamond character at the centre of the screen you enter

PRINT AT 1Ø,15;"◇"

where ◇ is obtained by putting the computer in graphics mode and pressing the D key.

Exercise
Using the RAM area USR "P" to USR "P" + 7 store the binary pattern corresponding to the user defined character shown in Fig. 7.4. Display the character and verify that it has been correctly stored.

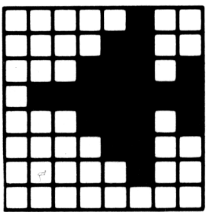
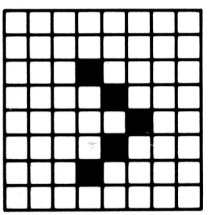


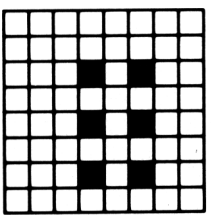
Fig. 7.4. User defined graphics character for aeroplane

The 8 × 8 pixel grid pattern enables 21818 (1.8446744E+19) different characters to be printed but you can only have 21 user defined characters. To create more characters however you can use the **OVER** 1 statement, which permits one character to overlay another character, but note that coincident 'inked' pixels are written as 'paper'.

As an example, let us consider the resulting character produced by overlaying > on <. The pixel grid definitions of > and < are given in Fig. 7.5 (a) and (b) respectively, and the resulting character is shown in Fig. 7.5 (c). You may display the newly created character by running three lines of program:

```
10 PRINT AT 10,15;"<"
15 OVER 1
20 PRINT AT 10,15;">"
```





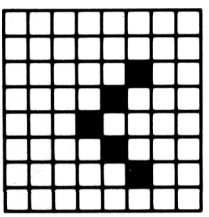


Fig. 7.5. Example of using **OVER** 1

The screen display

The paper on the screen is arranged as a 22 × 32 character grid as shown in Fig. 7.6. The rows are numbered from Ø to 21 starting from the top, and the columns are numbered from Ø to 31 starting from the left-hand side.

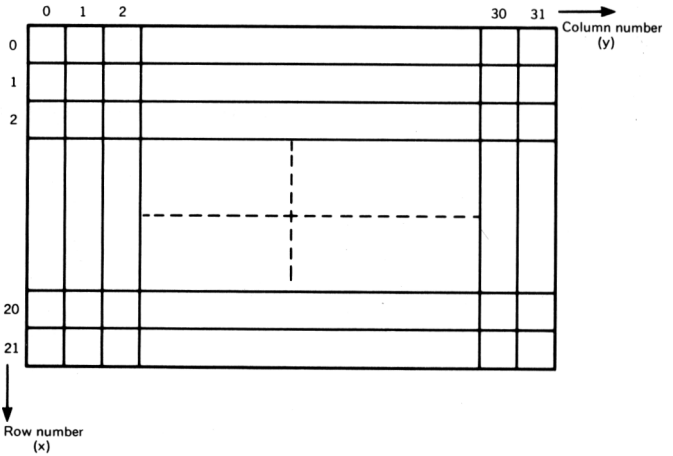


Fig. 7.6. Paper character-grid

You can print a character, or string of characters, at defined locations on the grid using the keyword **PRINT** followed by the **AT** pseudo-function in the format

PRINT AT row number, column number; "characters"

or alternatively in the format

PRINT AT row number, column number, **CHR$** code number

To move the **PRINT** position n columns along a specified row from the left-hand side of the grid you can use the pseudo-function **TAB**. This has the format **TAB** n, where n is the number (in the range Ø to 31) specifying the start position of the desired display.

The program below demonstrates use of **PRINT AT, TAB** and **CLS**. On running the program you will see the character display format shown in Fig. 7.7 appear at the bottom of the screen and then scroll.

```
 5 FOR n = 2Ø TO Ø STEP -1
1Ø PRINT AT n,9;"↑ UP & AWAY ↑ "
2Ø PRINT AT n + 1,9; CHR$ 94; TAB 21; CHR$ 94
25 PAUSE 2Ø
3Ø CLS
35 NEXT n
```

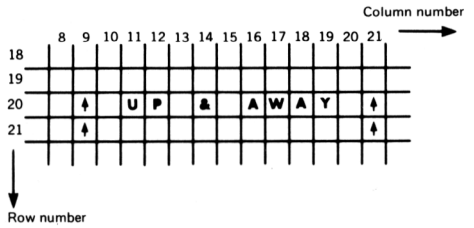


Fig. 7.7. Example of required character display

A character displayed in a defined location on the character grid can be read by using **SCREEN$** in the format

SCREEN$ row number, column number

For example, if you run

```
10 PRINT AT 4,4;"A"
20 PRINT CODE SCREEN$ (4,4)
```

you see that the ZX Spectrum displays the character A at the character position 4,4, and displays the code of A (65) at the beginning of the next line of the screen. Note however that you cannot use **SCREEN$** to read user defined characters.

For graphics applications the paper is arranged as a 176×256 grid of square picture elements, which are known as pixels. The pixel grid overlays the character grid, such that each square in the character grid contains sixty four pixels. The columns in the x-direction are numbered from 0 to 255 starting from the left-hand side, and the rows in the y-direction are numbered from 0 to 175 starting from the bottom. The pixel grid arrangement is shown in Fig. 7.8.

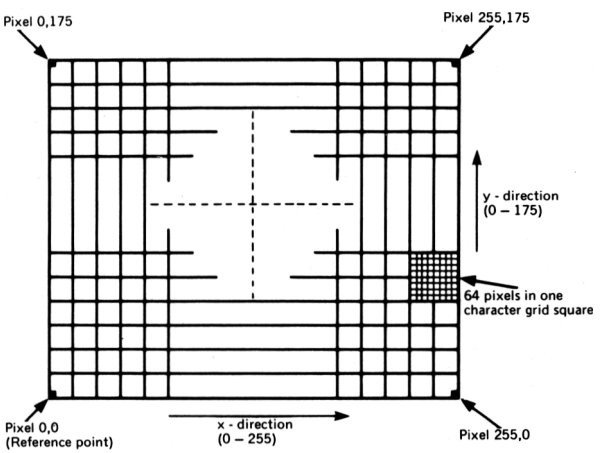


Fig. 7.8. Graphics pixel grid

The ZX Spectrum inks in a pixel at a defined location on the pixel grid using the keyword **PLOT**, followed by the x and y direction grid numbers. The format is

PLOT x-direction grid number, y-direction grid number

You may erase an inked in pixel from the paper by entering

PLOT x-direction grid number, y-direction grid number : **INVERSE** 1

To examine any pixel to determine whether it is ink or paper use the **POINT** function in the form

POINT (x-direction grid number, y-direction grid number)

and the result is 0 when the pixel is paper and 1 when the pixel is ink.

For example, the program line

15 **LET** x = **POINT** (200,100)

will set the variable x equal to 0 when pixel 200,100 is paper, otherwise x will be set equal to 1.

Drawing lines and circles

The ZX Spectrum has **DRAW** and **CIRCLE** statements which are used to plot straight lines, arcs and circles, as in the example shown in Plate 7.4.

To plot a straight line from a reference point you use the **DRAW** statement in the format

DRAW h,v

where h and v are the number of horizontal and vertical pixel displacements from the reference point. The sign of h must be positive if the horizontal displacement is to be to the right of the reference point, and must be negative for displacement to the left. The sign of v must be positive if the vertical displacement is to be upwards from the reference point and must be negative for downwards displacement. The reference point has the co-ordinates of the last pixel plotted by a preceding **PLOT, DRAW** or **CIRCLE** statement, and if these statements have not been used the reference is taken to be pixel 0,0. The reference point is reset to 0,0 by **NEW, RUN, CLS** or **CLEAR.**

Another form of **DRAW** statement is used to draw the arc of a circle. In this case the statement takes the form

DRAW h,v,a

Plate 1.1. The ZX Spectrum keyboard (courtesy of Sinclair Research)

Plate 1.2. The ZX Printer (courtesy of Sinclair Research)

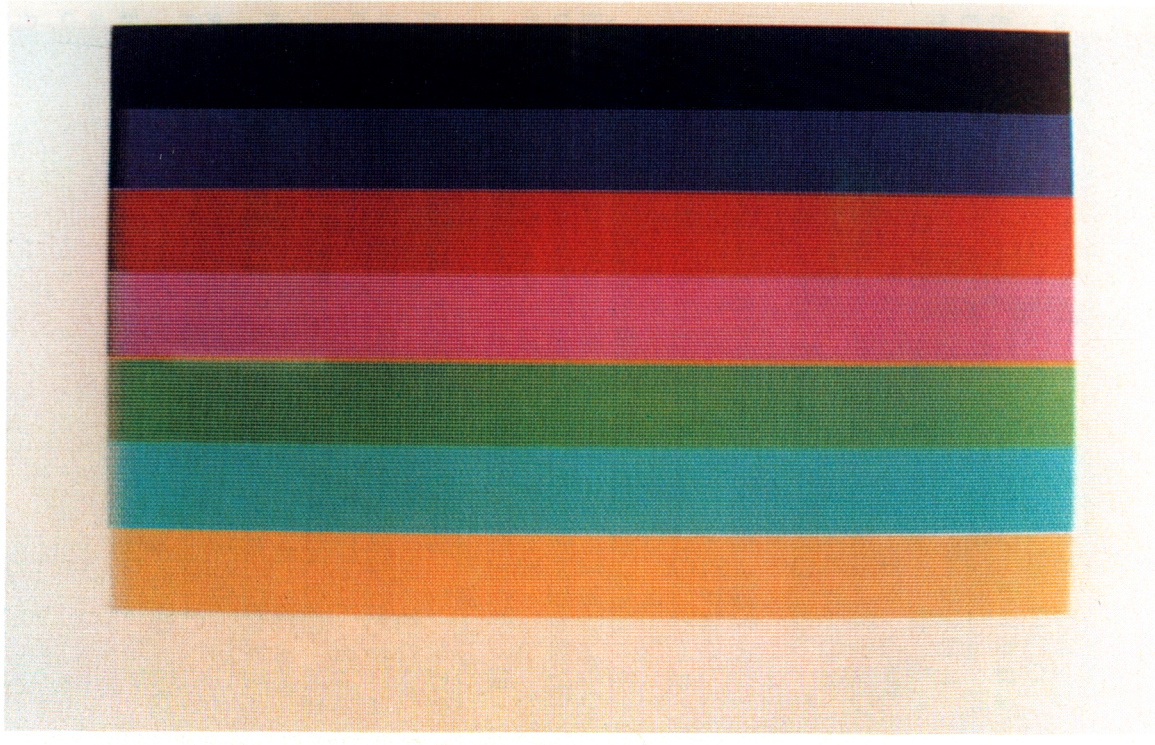


Plate 7.1. The eight available colours

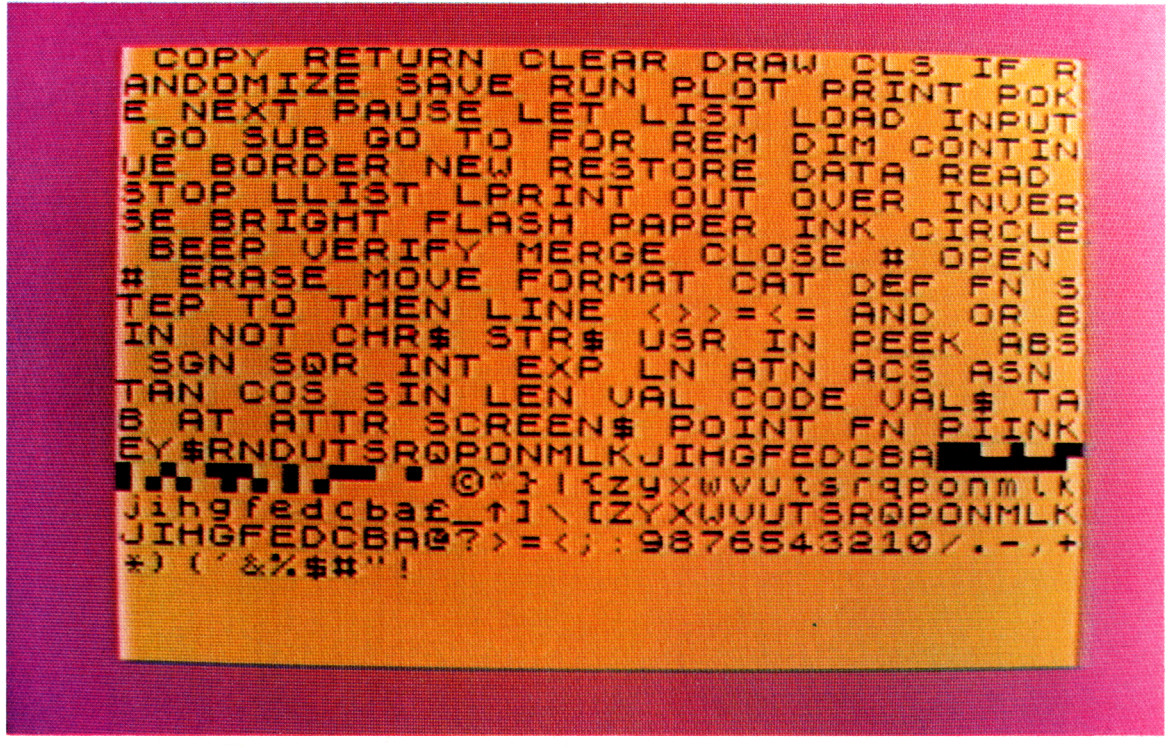


Plate 7.2. Display of the character set

where h and v are defined as above and 'a' is the angle of arc in radians. If 'a' has a positive value the arc is drawn in the counter-clockwise direction and when 'a' is negative the arc is drawn in the clockwise direction.

Exercise
Enter and run the following program to display a simple compass motif.

```
10 PLOT 196,125
20 DRAW -51,0
24 PLOT 150,125
25 DRAW 40,0,PI
26 DRAW -40,0,PI
30 PLOT 171,125
40 DRAW 0,27:DRAW 0,-51
60 PRINT AT 1,21;"N"
70 PRINT AT 2,21;"↑"
```

In the exercise above you will note that the circle in the motif has been drawn using two semicircular arcs (lines 25 and 26). However, there is an alternative way of drawing circles by using the **CIRCLE** statement.

The **CIRCLE** statement takes the form

CIRCLE x-direction pixel grid number, y-direction pixel grid number, radius in pixels

In the program in the exercise to draw the compass motif you could delete lines 24, 25 and 26 and insert

```
50 CIRCLE 171,125,20
```

to draw the circle part of the motif.

Note that if you intend to follow a **CIRCLE** statement with **DRAW** statements it is necessary to redefine the reference point using the **PLOT** statement.

It is possible to include, **INK, PAPER, BRIGHT, FLASH, OVER** and **INVERSE** in your **PLOT, DRAW** and **CIRCLE** statements, and these are placed immediately after the keyword and are delimited using commas and semicolons.

Exercise
Enter these two lines of program

```
10 DRAW PAPER 6,44,78
20 CIRCLE INK 4; FLASH 1 ; 100,78,15
```

and observe the displayed effect of the paper for the drawn line and circle.

Character-grid attributes

The screen character-grid consists of the paper character-grid (see Fig. 7.6) plus two additional rows, (rows 22 and 23) each with 32 character-grid elements (columns 0 to 31 inclusive). These additional rows are used to display lines of program as they are typed in, entered data, reports and prompts.

Each of the 768 (24 × 32) elements of the screen character-grid has an attribute which defines the colours of the paper and ink for the displayed character, and the associated display-mode for the character.

The attributes of each screen character-grid element are stored in predetermined random access memory locations in the address range 22528 to 23295 inclusive (Attributes File). Each screen character-grid element uses one byte of this area of memory to define its colours (see Plate 7.5) and display-mode (see the attribute byte format given in Fig. 7.9).

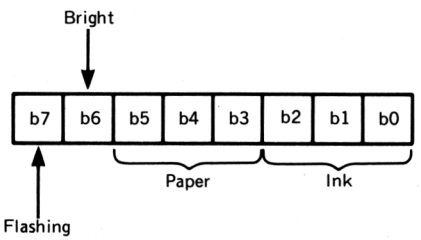


Fig. 7.9. Attribute byte format

The most significant bit, b7, of an attribute byte is used to define the flashing mode. If this bit is set to 1 the mode is flashing, otherwise 0 invokes a steady display. The next significant bit, b6, is used to define the bright mode. When this bit is set to 1 the mode is bright, and when it is set to 0 the mode is normal. Bits b5, b4 and b3 are used to define the colour of the paper of the screen character-grid element and bits b2, b1 and b0 define the ink colour of the character element.

The decimal equivalent of the flashing mode bit, b7, when set to 1 is 128, and 0 otherwise. The decimal equivalent of the bright mode bit, b6, when set to 1 is 64, and 0 otherwise. The paper colour bits have a decimal equivalent of eight times the colour key number, while the ink colour bits have a decimal equivalent of the colour key number. The attribute therefore can have a decimal equivalent value 0 to 255 that is determined by using

Attribute decimal value = (b7 × 128) + (b6 × 64) + (paper key number × 8) + ink key number

The attribute of any screen character-grid element can be examined using the function **ATTR** in the format

ATTR (line number, column number)

For example, if after entering **NEW** you run the lines of program

```
1Ø PAPER 6 : INK 3
2Ø PRINT ATTR (Ø,Ø)
```

the Spectrum displays 51, indicating that the paper colour number is 6, the ink number is 3 and the display mode is steady and normal.

To set the attribute for a screen character-grid element you can use the **POKE** statement in the format

POKE attribute file address, attribute decimal value

where the attribute file address is related to the row and column numbers for the screen character-grid by

Attribute file address = 22527 + (column number + 1) + ((row number)*32)

Exercise
Verify that when you enter

POKE 23231,2Ø7

the bottom right-hand screen character-grid element of the paper flashes bright with paper 1 (blue) and ink 7 (white). Verify that if you now enter

PRINT ATTR (21,31)

the ZX Spectrum responds with the correct attribute.

More about the screen

Earlier we saw that you can display characters and pixels on the screen, and Figs. 7.6 and 7.8 define the corresponding grids. The data bytes which define displayed characters and pixels are stored in the area of random access memory known as the display file. The display file has the address range 16384 to 22527 inclusive, but the eight bytes of a character are not stored sequentially and, furthermore, the computer does not sequentially write the 192 lines of the display (24 character rows × 8 bytes/character = 192 lines).

The character screen grid shown in Fig. 7.6, plus the two character rows used for reports, prompts, etc, is arranged as three sections of 32 columns and 8 rows, as shown in Fig. 7.10. The characters are written onto the screen, starting at character row 0, screen section 0, and the

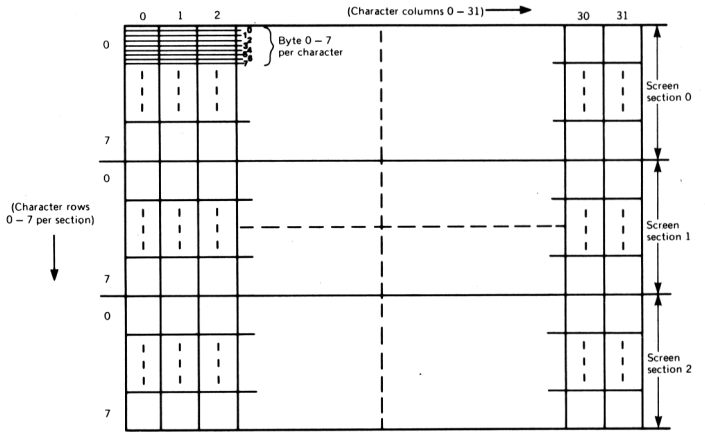


Fig. 7.10. Screen display format

computer sequentially outputs byte 0 for each of the 32 characters in row 0, starting at column 0. Then byte 0 for all 32 characters in row 1, screen section 0, is sequentially displayed, followed by byte 0 for the remaining rows in screen section 0. Then byte 1 for all characters of section 0 is similarly displayed, followed by byte 2, and so on until all characters in screen section 0 have been displayed. Screen section 1 is displayed in a similar manner and finally screen section 2 is completed. You can observe this rather complicated process by using

```
2Ø FOR n = 16384 TO 22527
3Ø POKE n,255
4Ø NEXT n
```

To calculate the address of any display file byte you can use

Display file address = 16384 + character column number + (32* character row number) + (256* character byte number) + (2048 × screen section number)

where character column number, character row number, character byte number and screen section number are as shown in Fig. 7.10. For example, the address of byte 6, character row 5, column 27, screen section 2, is

16384 + 27 + (32*5) + (256*6) + (2048*2) = 22203

and if you

```
POKE 222Ø3,255
```

you will see the eight pixels inked-in, in the correct position on the

screen. Recall that 255 = 11111111, that is one byte defines eight pixels.

The ability to be able to **POKE** data bytes into the display file in the required locations is a feature which is especially useful when programming using machine code.

Moving graphics

We have seen how to display a graphic character in a specified character-grid square using **PRINT AT** x,y; "character", where x and y represent the row and column numbers respectively. If we keep x constant and increment y from Ø to 31, then the character will be displayed in every column of row x. For example, the program

```
 5 FOR y = Ø TO 31
15 PRINT AT 1Ø, y;"*"
25 NEXT y
```

displays a line of asterisks in row 1Ø. When you run this program you will note that the graphics character moves from left to right, but leaves an image along its path. The trailing image can be removed by changing line 15 to

```
15 PRINT AT 1Ø, y;" *"
```

that is, one space has been included immediately in front of the asterisk. Note that the program now displays a moving asterisk starting from column 1 and finishing in column 0 after moving through the columns.

To reverse the direction of movement, say when the asterisk reaches the right-hand side of the screen, the program may be written to detect when y equals a specified upper limit, and then y may be decremented to reverse the direction of movement. To avoid a trailing image created by the reversal of direction a space should be included after the asterisk.

If you run the following program you will see that the asterisk bounces back and forth across the screen.

```
15 FOR y = Ø TO 3Ø
25 PRINT AT 1Ø,y;" *"
3Ø IF y = 3Ø THEN GO TO 45
35 NEXT y
45 FOR y = 3Ø TO Ø STEP -1
55 PRINT AT 1Ø,y;"* "
6Ø IF y = Ø THEN GO TO 15
65 NEXT y
```

To achieve vertical movement of the asterisk, the y parameter is kept constant and the x parameter is varied according to the movement required. Of course, **PRINT AT** statements must be included to display spaces above and below the asterisk to remove the trailing images. Furthermore, in some applications it may be desirable to control movement of the graphics from the keyboard. To investigate vertical movement of the asterisk controlled from the keyboard we suggest that you run the following program. By pressing the U or D key the asterisk will move up or down respectively between the upper and lower limits of the screen.

```
 15 LET x = 10
 25 LET y = 15
 35 LET K$ = INKEY$
 45 IF K$ = "U" THEN LET x = x - 1
 50 IF x <0 THEN LET x = 0
 55 IF K$ = "D" THEN LET x = x + 1
 60 IF x >20 THEN LET x = 20
 75 PRINT AT x - 1,y;" "
 85 PRINT AT x,y;"*"
 95 PRINT AT x + 1,y;" "
115 GO TO 35
```

To study the graphics capability of the ZX Spectrum further, let us consider how to achieve movement of the displayed asterisk on an approximately circular path on the screen. In this case the x and y parameters must be evaluated in turn for each required display position. In the following program the co-ordinates of the character-grid square (which lies on the circumference of the circle) are evaluated in lines 45 and 55, and the asterisk is then printed at the required position on the screen. Note how the two **LET** statements in lines 36 and 37 are used to save the 'old display position', which is subsequently used in line 58 to wipe out the trailing image. Run the program and see how the asterisk moves around the screen.

```
  5 LET H = 0
  7 LET U = 0
 35 FOR A = 0 TO (2*PI) STEP PI/8
 36 LET M = H
 37 LET N = U
 45 LET H = 15 + INT (8*COS A)
 55 LET U = 10 + INT (8*SIN A)
 58 PRINT AT N,M;" "
 65 PRINT AT U,H;"*"
 75 NEXT A
 95 GO TO 35
```

A simple video game using some of the above principles is given in Program 5 in Appendix A.

You can also produce movement by inking and erasing pixels. For example, the following program can be used to display a single pixel which continuously travels horizontally back and forth across the screen.

```
  3 BORDER 2: PAPER 6: INK Ø
  5 LET y = 85
 1Ø FOR x = Ø TO 255
 15 INVERSE Ø : PLOT x,y
 18 PAUSE 2
 2Ø INVERSE 1 : PLOT x,y
 35 NEXT x
 45 FOR x = 255 TO Ø STEP -1
 55 INVERSE Ø : PLOT x,y
 65 PAUSE 2
 75 INVERSE 1 : PLOT x,y
 95 IF x = Ø THEN GO TO 5
1ØØ NEXT x
```

Note that the pixel is inked in using the **INVERSE** Ø : **PLOT** x,y statements (lines 15 and 55) and it is erased using the **INVERSE** 1: **PLOT** x, y statements (see lines 20 and 75). The **PAUSE** statements in lines 18 and 65 are necessary to prevent unwanted erasure - verify this by removing these two lines and observe the change in the display.

Exercise

Run the following program and see how the pixel moves vertically up and down the screen.

```
  3 BORDER 3: PAPER 4: INK 2
  5 LET x = 128
 1Ø FOR y = Ø TO 175
 15 INVERSE Ø : PLOT x,y
 18 PAUSE 2
 2Ø INVERSE 1 : PLOT x,y
 35 NEXT y
 45 FOR y = 175 TO Ø STEP -1
 55 INVERSE Ø : PLOT x,y
 65 PAUSE 2
 75 INVERSE 1 : PLOT x,y
 95 IF y = Ø THEN GO TO 5
1ØØ NEXT y
```

Concluding remarks

In this chapter we have studied the colour graphics capabilities of the ZX Spectrum, and seen how it stores and outputs graphic characters to the screen. We have also dealt with user defined graphics, moving graphics, and how to draw circles and lines. All of these features should be useful in developing programs for colour graphics applications.

8

Sounds

Introduction

The ZX Spectrum can generate sound and this of course is noticed every time you press a key. If you find that when you press a key that the sound level is too low you can increase it by POKEing into the system variable PIP, at address 236Ø9, using

POKE 236Ø9, Integer in the range Ø to 255

We suggest that you investigate this using several data values POKEd into PIP (see Appendix C).

Let us now consider how to program the sound generator to produce notes having specified duration and pitch.

Sound generation

You can program your ZX Spectrum to generate electronically created sound of predetermined duration and pitch by using the **BEEP** statement in the form

BEEP duration, pitch

where duration and pitch may be numerical expressions or numerical values, which may be integer or non integer. The duration must be expressed in seconds, and pitch, the frequency of the generated note, is expressed as the number of semitones relative to middle C, ie the pitch value is zero for the reference frequency of 261.6 Hz in the equally tempered scale (agreed at the London International Conference, May 1939). In this equally tempered scale a semitone is the interval between two adjacent frequencies, such that a frequency f_n is related to the next highest semitone f_{n+1} by

$$f_{n+1} = \sqrt[12]{2}\, f_n = 1.0594631\, f_n$$

For convenience, when using the **BEEP** statement, the semitone frequency is identified as being an integer number of semitones relative to middle C. Positive pitch numbers correspond to semitones higher than middle C and negative pitch numbers correspond to semitones lower than middle C. Table 8.1 lists 27 semitones, the

Table 8.1. Equally tempered scale of notes

Musical note	*ZX Spectrum pitch number*	*Frequency of note (Hz)*
C	−12	130.8
C♯, D♭	−11	138.6
D	−10	146.8
D♯, E♭	− 9	155.6
E	− 8	164.8
F	− 7	174.6
F♯, G♭	− 6	185.0
G	− 5	196.0
G♯, A♭	− 4	207.7
A	− 3	220.0
A♯, B♭	− 2	233.1
B	− 1	246.9
C	0	261.6
C♯, D♭	1	277.2
D	2	293.7
D♯, E♭	3	311.1
E	4	329.6
F	5	349.2
F♯, G♭	6	370.0
G	7	392.0
G♯, A♭	8	415.3
A	9	440.0
A♯, B♭	10	466.2
B	11	493.9
C	12	523.2
C♯, D♭	13	554.4
D	14	587.4

corresponding ZX Spectrum pitch number and the actual frequency of the note. From this table you will see that after 12 semitones, ie an octave, the frequency of a note doubles.

The following simple program will generate the scale of two octaves, starting one octave below middle C, and moving up in semitone steps to finish one octave above middle C.

```
 3 PRINT "Input 'duration' of BEEP"
 4 INPUT y
 5 FOR x = -12 TO 12
15 BEEP y,x
25 NEXT x
```

We suggest you run this program with a duration value in the range 0.0015 seconds to 2 seconds.

You may find it interesting to vary the pitch according to a mathematical expresssion. For example the program listing below uses the expression (sin x + cos x)x to define the pitch value.

```
 3 PRINT "Input 'duration' of BEEP"
 4 INPUT y
 5 FOR x = -12 TO 12
15 BEEP y, (SIN x + COS x)*x
25 NEXT x
30 GO TO 5
```

We suggest you run this program with a duration value of 0.035 seconds. To exit from the program loop simply press **CAPS SHIFT** and **BREAK.**

To introduce a *rest* into the generated musical note sequence you may use the **PAUSE** statement, which takes the form

PAUSE n

where n is an integer in the range 0 to 65535. If n = 0 the pause lasts until any key is pressed, otherwise if n ≠ 0 the pause lasts n/50 seconds. Consequently the maximum pause you can obtain using a single **PAUSE** statement is 21.845 minutes. (In North America the pause lasts n/60 seconds giving a maximum pause of 18.204167 minutes.)

You may find it interesting to insert

18 **PAUSE ABS** x + 1

in the above program, with a duration value of 0.035 seconds.

Exercise

Try various mathematical expressions to define the pitch value in the **BEEP** statement in line 15 of the above program. Also investigate the effect of varying the duration value.

In undertaking the exercise above you will create your own sound sequences. However, you may wish to translate published music so that the ZX Spectrum can be programmed to play tunes. To do this you need to understand the rudiments of music and in the following section a brief introduction is presented. If you can read music you will probably wish to move on to the illustrative example.

Rudiments of music

Music for the piano, violin, flute, guitar, etc, is often written in treble or G clef. The clef is placed on the left-hand side of the five lines and

four spaces on which the music is written (this is called the staff). The G clef encircles the second line of the staff and this establishes the note G on this line, and from this reference the other lines and spaces are named. In addition notes are written on and between short lines above and below the staff, and these lines are called leger lines (see Fig. 8.1).

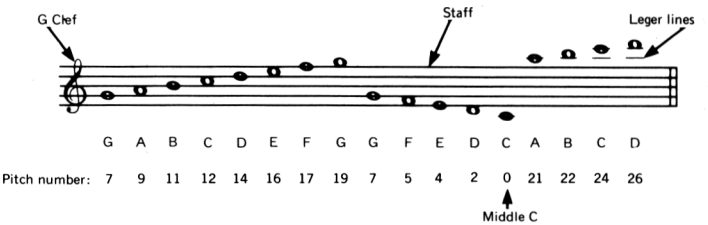


Fig. 8.1. Musical notes with corresponding ZX Spectrum pitch numbers

The staff is divided by barlines into bars, which are equal in time value, and this is specified by the time signature (fractional number) placed at the beginning of the music. The time signature indicates the number of notes of equal value in each bar, where the upper figure gives the number of beats in a bar, and the lower figure indicates the type of note that has one beat. For example, $\frac{4}{4}$ (or C) means that there are 4 beats in a bar and that one beat is a quarter note (a crotchet).

There are several types of notes, each one representing a certain time value. In Fig. 8.2 are shown the semibreve, minim, crotchet and

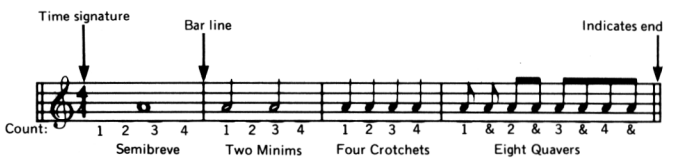


Fig. 8.2. Common musical notes

quaver with a $\frac{4}{4}$ time signature. The number of beats to the bar for the semibreve is four, two for each minim, one for each crotchet and one for each group of two quavers. The fermata symbol, 𝄐, written over a note indicates that the note should be held for as long as possible. To indicate that the pitch of a note is to be changed, appropriate symbols are placed before the note:

(i) the sharp symbol, ♯, indicates that the pitch should be raised by one semitone;
(ii) the flat symbol, ♭, indicates that the pitch should be lowered by one semitone;

(iii) the natural symbol, ♮, cancels the previous sharp or flat;
(iv) the double sharp symbol, x, raises the pitch by a full tone; and
(v) the double flat symbol, 𝄫, lowers the pitch by a full tone.

A rest indicates a pause for the value of the note after which it is named. In Fig. 8.3 we show the semibreve, minim, crotchet, quaver and semiquaver rest.

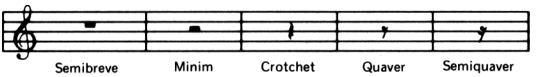


Fig. 8.3. Musical rests

A dot placed after a note or a rest increases its value to one and a half. Some examples are shown in Fig. 8.4(a). Two dots placed after and before double bar lines indicate that the music between these signs is to be repeated, as in Fig. 8.4(b). The Del Segno (D.S.) means return to the sign 𝄋 and play to the end, and Da Capo (D.C.) means return to the beginning and play to the end.

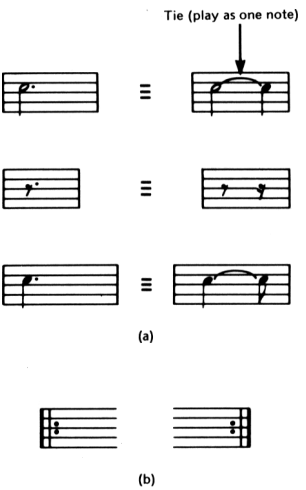


Fig. 8.4. (a) Dot notation for notes and rests, (b) dot notation to indicate repeat

Every key has two names, one when the music is major and the other when it is a minor. The key for the music is indicated by the number of sharps or flats at the beginning of each piece of music (see Fig. 8.5).

Let us now use some of these rudiments in the following example.

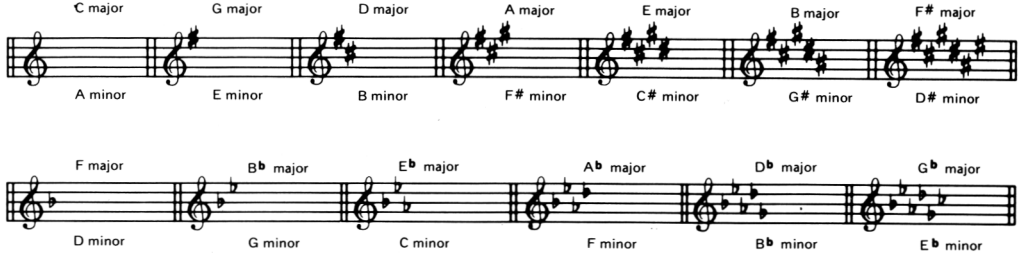


Fig. 8.5. Musical keys

Illustrative example

In Fig. 8.6 we show the music and corresponding ZX Spectrum pitch numbers of a well-known tune. The following program implements

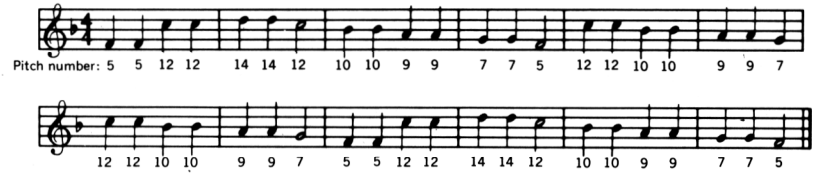


Fig. 8.6. Twinkle, Twinkle, Little Star

the required sound generation. You will note that the duration value is entered by the user and after entering this value the tune is played. We suggest you try a duration value of 0.3 seconds and then try other values.

```
 5 PRINT "Twinkle, Twinkle, Little Star"
1Ø INPUT d:PRINT d
15 BEEP d,5:BEEP d,5:BEEP d,12:BEEP d,12
2Ø BEEP d,14:BEEP d,14:BEEP 2*d,12
25 BEEP d,1Ø:BEEP d,1Ø:BEEP d,9:BEEP d,9
3Ø BEEP d,7:BEEP d,7:BEEP 2*d,5
35 BEEP d,12:BEEP d,12:BEEP d,1Ø:BEEP d,1Ø
4Ø BEEP d,9:BEEP d,9:BEEP 2*d,7
44 BEEP d,12:BEEP d,12:BEEP d,1Ø:BEEP d,1Ø
47 BEEP d,9:BEEP d,9:BEEP 2*d,7
5Ø BEEP d,5:BEEP d,5:BEEP d,12:BEEP d,12
55 BEEP d,14:BEEP d,14:BEEP 2*d,12
6Ø BEEP d,1Ø:BEEP d,1Ø:BEEP d,9:BEEP d,9
65 BEEP d,7:BEEP d,7:BEEP 2*d,5
```

Audio amplifier

The audio signal from the internal sound generator of the ZX Spectrum appears on the MIC and EAR sockets. So, by using a 3.5 mm jack plug, you can connect this signal to the input of an audio amplifier to obtain amplified sound. An appropriate audio amplifier circuit is given in Fig. 8.7. The circuit has a volume control (10 kΩ potentiometer) and uses an inexpensive audio amplifier chip, type LM380. The pin assignment of the LM380 is given in Fig. 8.8.

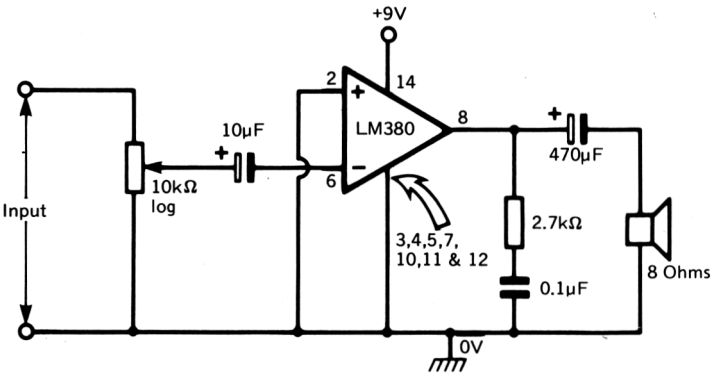


Fig. 8.7. Audio amplifier circuit

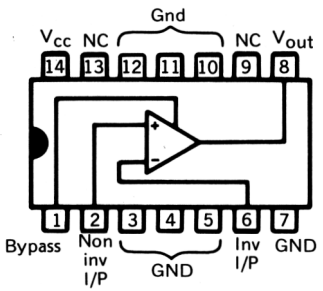


Fig. 8.8. Pin assignment of the LM380 audio amplifier

Concluding remarks

In this chapter we have introduced the basic concepts of sound generation using the **BEEP** statement, and have shown how it is possible to translate music into the form required by the computer. This provides you with the facility to include sound generating statements as part of your program. For an example see Program 5 in Appendix A.

We have only considered sound generation using the **BEEP** statement, but it is possible to generate sound using machine-code and we consider this aspect in Chapter 10.

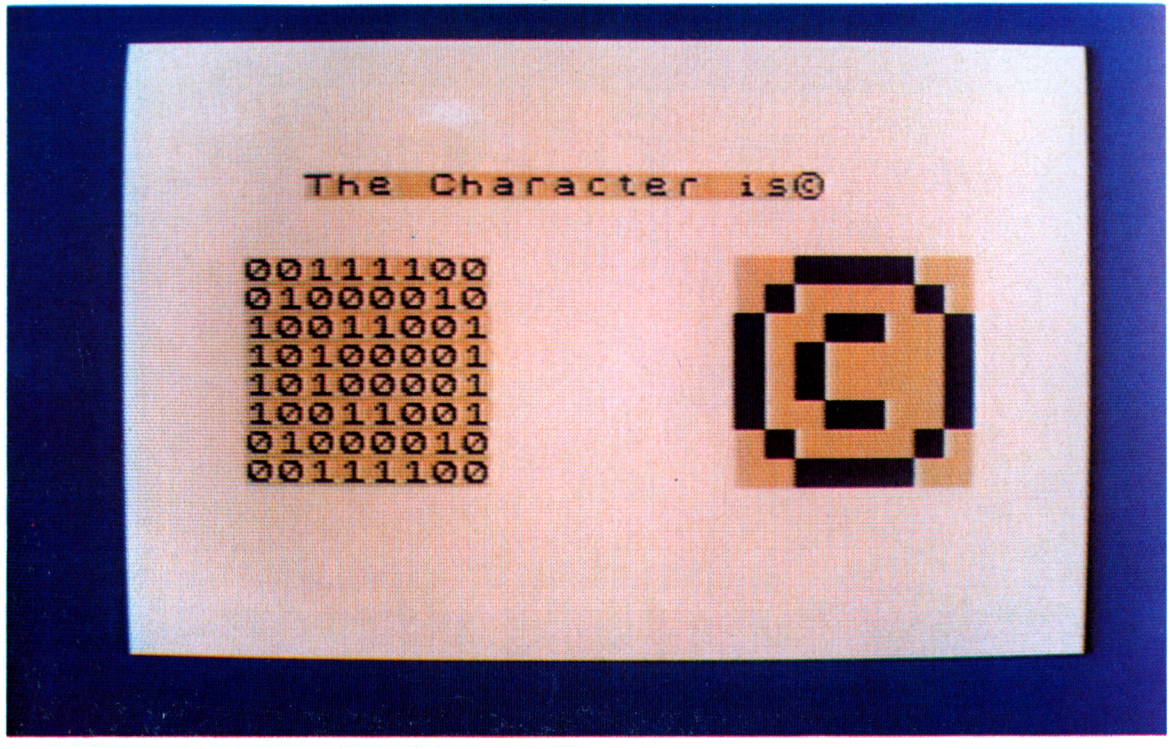


Plate 7.3. Example of output of Program 3, Appendix A

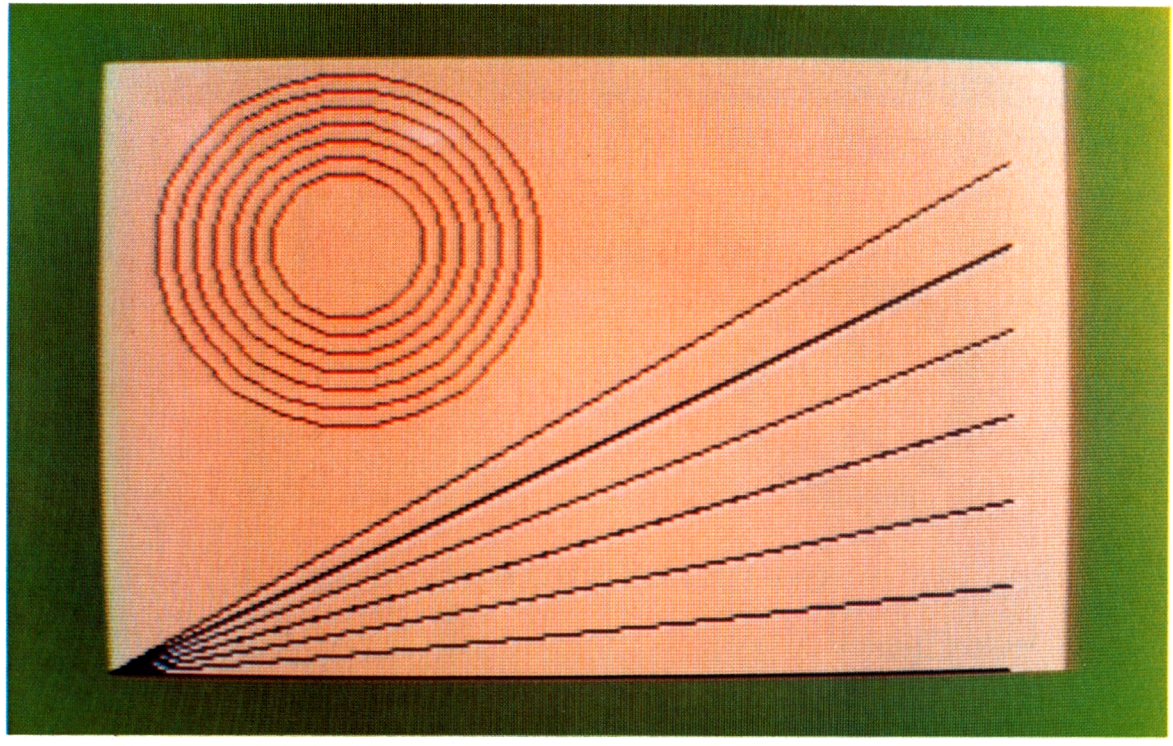


Plate 7.4. Example of drawing lines and circles

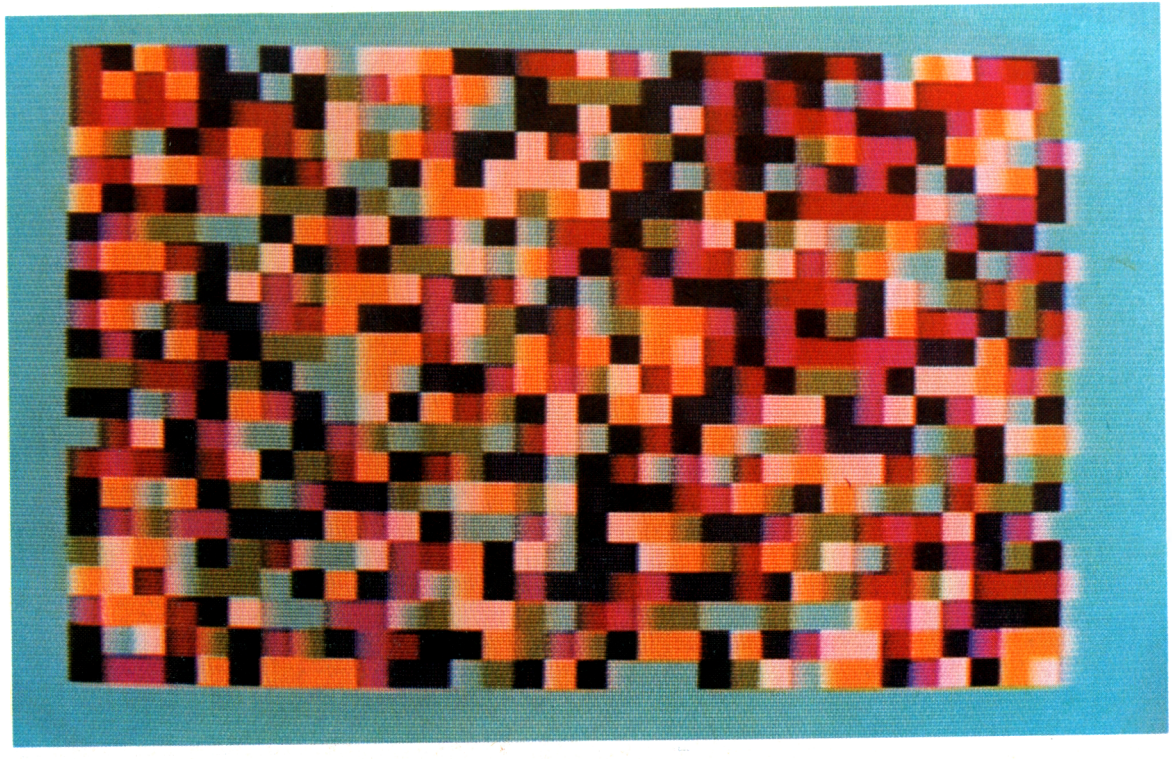


Plate 7.5. Character grid with randomly set colour attributes

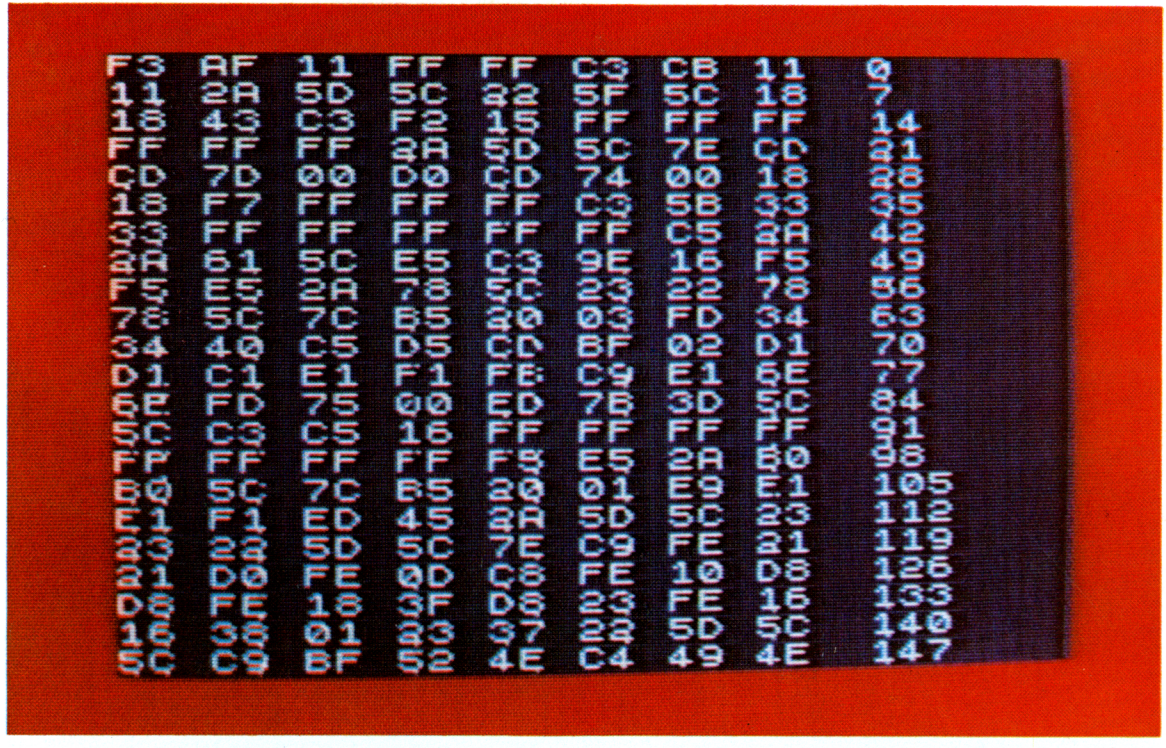


Plate A.4. Display produced by ROM dump program

9

Hardware

Introduction

The ZX Spectrum computer contains a number of interconnected integrated circuits which form part of the hardware of the system. A simple block diagram is shown in Fig. 9.1. The computer system consists of a microprocessor unit (MPU), memory for storing instructions and data, and some input/output capability.

The microprocessor is the heart of the system, and its role is to execute arithmetic or logical operations in accordance with a program of instructions. Two types of memory device are used: random access memory (RAM) for storing your program, data, system variables, etc; and read only memory (ROM) for storing the 16K (16 × 1024 bytes of memory) Basic interpreter and operating system.

Information is input to the microcomputer by using the keyboard, the cassette recorder, the microdrive and a user interface. Output information from the microcomputer may be directed to the cassette recorder, the printer, a TV via the PAL encoder and UHF/VHF modulator, the microdrive, the user interface, and the audio circuit and a loudspeaker.

To ensure that the system operates correctly under user and program control extensive timing and control circuitry is necessary. It is evident that the Ferranti Uncommitted Logic Array, ULA, (type ULA5C102E in the 16K Spectrum and type ULA5C112E in the 48K Spectrum) has been designed to implement necessary timing and control of the hardware. Although full details of the ULA chip are not provided and are not readily available in published literature, it is apparent that the device is used for:

(a) chip select and address decode for RAM and ROM;

(b) timing and control of the Z80A microprocessor;

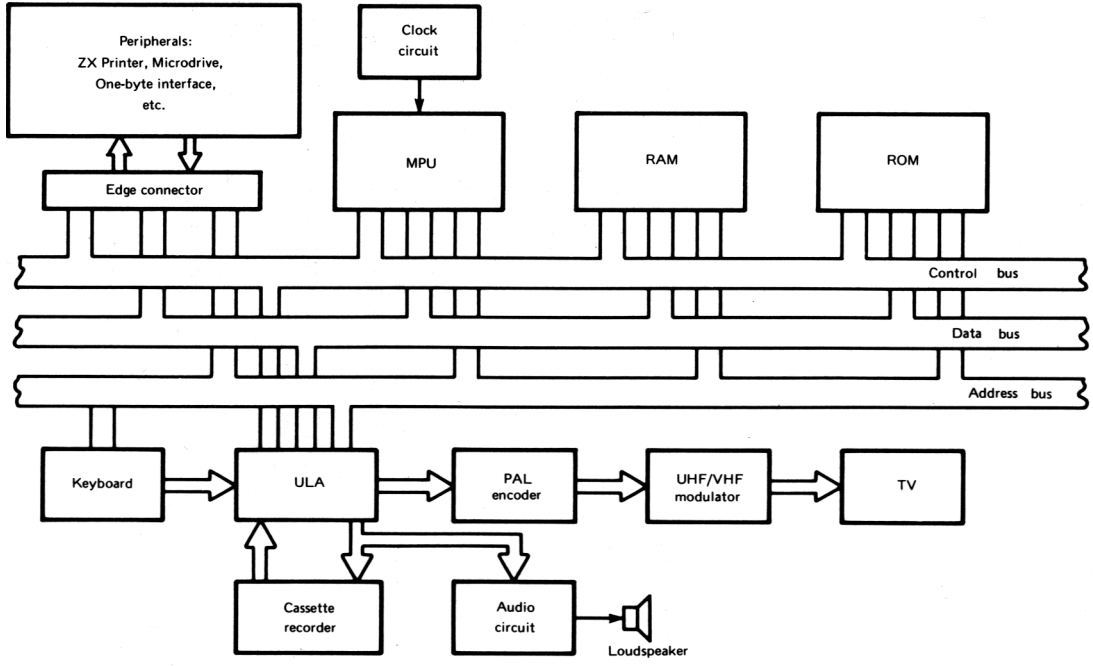


Fig. 9.1. Simple block diagram of the ZX Spectrum

(c) sensing keyboard operation;
(d) control signal conditioning and transfer of input and output information.

We shall now consider some digital system concepts which will assist you in understanding the ZX Spectrum's Z80A microprocessor. This knowledge of digital systems will also be useful when you wish to know how external hardware may be interfaced to the computer.

Flip-flops, flags, registers and counters

Flip-flops are important elements in microcomputers and may be used as memory cells to store binary bits (logic 0 or logic 1).

The simple Set-Reset (SR) flip-flop can be implemented using NAND or NOR gates. The logic symbol and basic NAND gate implementation are shown in Fig. 9.2, and we see that when the Set-input is changed from logic 1 to logic 0, the Q-output goes to logic 1 and correspondingly $\overline{Q}$ goes to 0. The outputs stay at these logic levels even after the Set-input returns to logic 1, and will only subsequently give a logic 0 at the Q-output and logic 1 at the $\overline{Q}$-output after the application of a logic 0 to the Reset-input. This type of flip-flop has ambiguous output states when the S and R inputs are set at 0 and, by design, this state is avoided in practice.

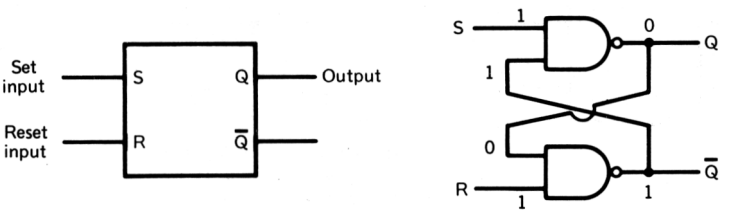


Fig. 9.2. Symbol and NAND gate implementation of Set-Reset flip-flop

This flip-flop is often used in applications requiring only the availability of the Q-output and in such cases the flip-flop is referred to as a status flag, and the Q-output is called the flag output.

Flags are used for indicating the occurrence of events taking place within a microprocessor. For example, if the execution of an arithmetic operation results in the answer being zero, a flag is set in the Z80A microprocessor. If the result is not zero the flag will be reset. The flag in this case is referred to as the Z-flag and is one of the status flags contained in a group within the microprocessor called the Flag Register.

The D-type flip-flop is similar to the Set-Reset flip-flop but it has one

data input (D), a clock input (C) and two outputs Q and $\overline{Q}$. The binary value (data) to be stored is applied to the D-input and on receipt of the positive-going edge of a clock pulse the binary value on the D-input is stored in the flip-flop and appears at the Q-output. The inverted (negated) value of the D-input appears at the $\overline{Q}$-output. The symbol for this type of flip-flop is shown in Fig. 9.3.

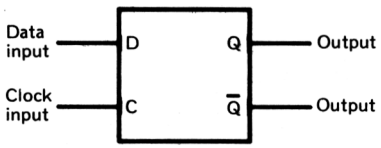
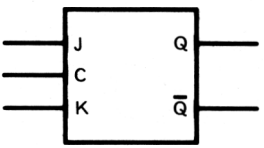


Fig. 9.3. Symbol for a D-type flip-flop

The clocked J-K flip-flop is similar to the Set-Reset flip-flop, because it has two inputs, J and K, and these control the state of the flip-flop in conjunction with an applied clock pulse. The input condition J = K = 1 causes the Q-output to change state (toggle) on receipt of each negative-going edge of the clock pulse. The symbol and function table for a J-K flip-flop are given in Fig. 9.4.



Inputs		Output
J	K	Q_{n+1}
0	0	Q_n
1	0	1
0	1	0
1	1	$\overline{Q}_n$

Where Q_n = state of flip-flop before clock pulse

Q_{n+1} = state of flip-flop after clock pulse

Fig. 9.4. Symbol and function table for a clocked J-K flip-flop

We have seen that a flip-flop can store one bit of information. When flip-flops are organised to store a binary word the arrangement is referred to as a register. If the data is set into and read out of all flip-flops simultaneously, the register is a parallel-in, parallel-out register, whereas if the data is entered one bit at a time and read out of all flip-flops simultaneously, the register is a serial-in, parallel-out register.

An eight-bit parallel-in, parallel-out register, constructed using D-type flip-flops, is shown in Fig. 9.5. When the write line to the register is pulsed positively the eight-bit binary input word is stored in the flip-flops. A two-input AND gate is included; one input to each gate is the Q-output from the corresponding D-type flip-flop, the

other input to the gate is connected to the read line. Consequently the true output word will only appear on the output data lines when a read signal is applied.

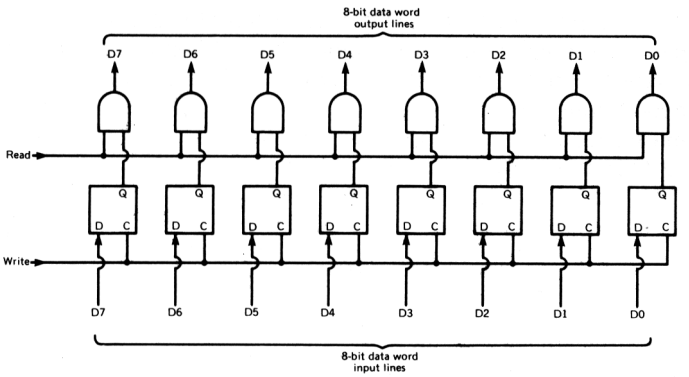


Fig. 9.5. An eight-bit parallel register

When it is necessary to manipulate data in serial form a shift register is used. Figure 9.6 illustrates an eight-bit shift register implemented using eight J-K flip-flops. The first bit of the data is transferred into the leftmost flip-flop on receipt of the first negative-going edge of the clock pulse. On each successive negative-going edge of the clock pulse each data-bit is shifted right into the next flip-flop, and a new data-bit enters the leftmost flip-flop. It requires eight clock pulses to load an eight-bit serial word into this register. If the output of all eight flip-flops are read after eight clock pulses have been applied, the serial word is available in a parallel form. This is a method of serial to parallel conversion.

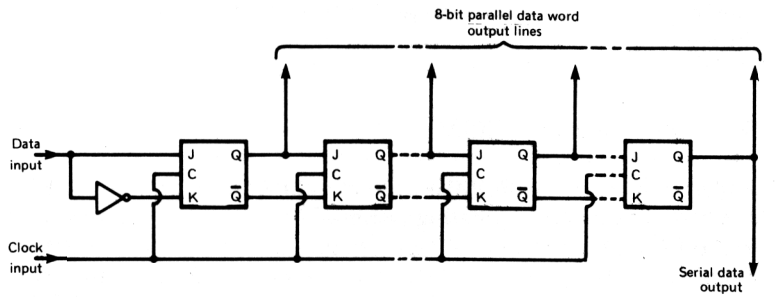


Fig. 9.6. Eight-bit shift register constructed using J-K flip-flops

It is worth noting that the Z80A microprocessor in your ZX Spectrum contains several eight-bit and 16-bit special-purpose and general-purpose registers, and you will meet these in Chapter 10.

Flip-flops may be connected together to form different types of counters. The J-K flip-flop is especially suited for this application because it has toggle capability when J and K are set to 1. Figure 9.7 shows how four J-K flip-flops may be connected to form a 0000 to 1111 binary ripple-up counter. We can consider the output of each flip-flop to have a weighted binary value. The first (left-hand) flip-flop has a weight of 1, the second has a weight of 2, the third a weight of 4 and the right-hand flip-flop has a weight of 8. This counter continues to count on successive input pulses and on the next clock pulse after reaching 1111 it *rolls over* to 0000.

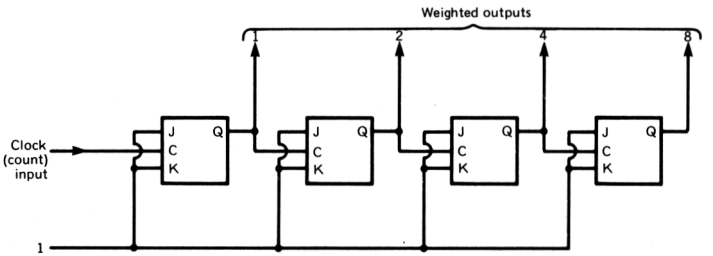


Fig. 9.7. Four-stage binary ripple-up counter

The addition of extra flip-flops increases the maximum possible count value. We can see that a binary ripple-up counter constructed using N flip-flops will be able to count 2^N-1 pulses before rolling over.

The Z80A contains an accessible 16-bit Program Counter which contains the address of the next instruction to be fetched from memory. The counter may be preloaded and its content is changed during program execution.

Read only memory (ROM)

A read only memory (ROM) is used in your ZX Spectrum to store the Basic interpreter and operating system. This ROM stores information in the form of binary words (see Chapter 2) with each binary word held in an addressable memory location, and once programmed its contents cannot be changed. Furthermore, because the ROM is static it does not need to be refreshed. Also note that on removal of the power supply the content of the ROM is not lost and it is therefore said to be *non-volatile*.

The ROM can be considered to be an array of single-bit memory

cells arranged in groups of eight bits (bytes) forming addressable memory locations as shown in Fig. 9.8. Each memory cell is programmed by the manufacturer during the fabrication (manufacturing) process in accordance with the customer's

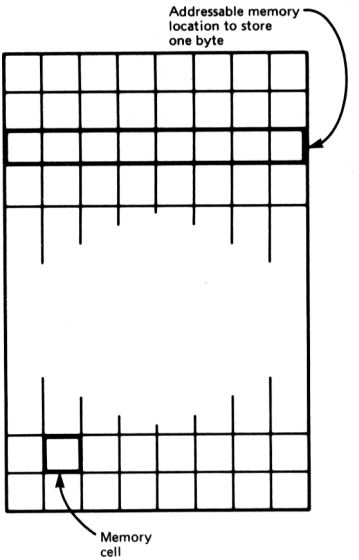


Fig. 9.8. Memory cell array

requirements to store permanently logic 0 or logic 1. The chip also contains an address decoder, and some control and interface logic. The device is a 16384 word by eight-bit mask programmable ROM, operating from a single 5 V power supply, and is byte organised and designed for use in bus-organised systems. A block diagram of the HN61312P ROM is given in Fig. 9.9(a), and the pin assignment is shown in Fig. 9.9(b).

There are eight unidirectional tri-state data output bus connections, D0 to D7, which are used to output on to the data bus the content of an addressed memory location during a read operation. It has fourteen address bus connections, A0 to A13 and therefore it is possible to access (maximum access time 250 ns) any one of the 16384(2^{14}) words of memory. The chip select input, CS, when active (active level is user defined) enables the address decode logic and brings the chip from its power-down mode (5 μW typically) to its normal operating mode (50 mW typically). The two output enable inputs OEO and OE1 (active levels are user defined) are used to enable the TTL-compatible tri-state output buffers.

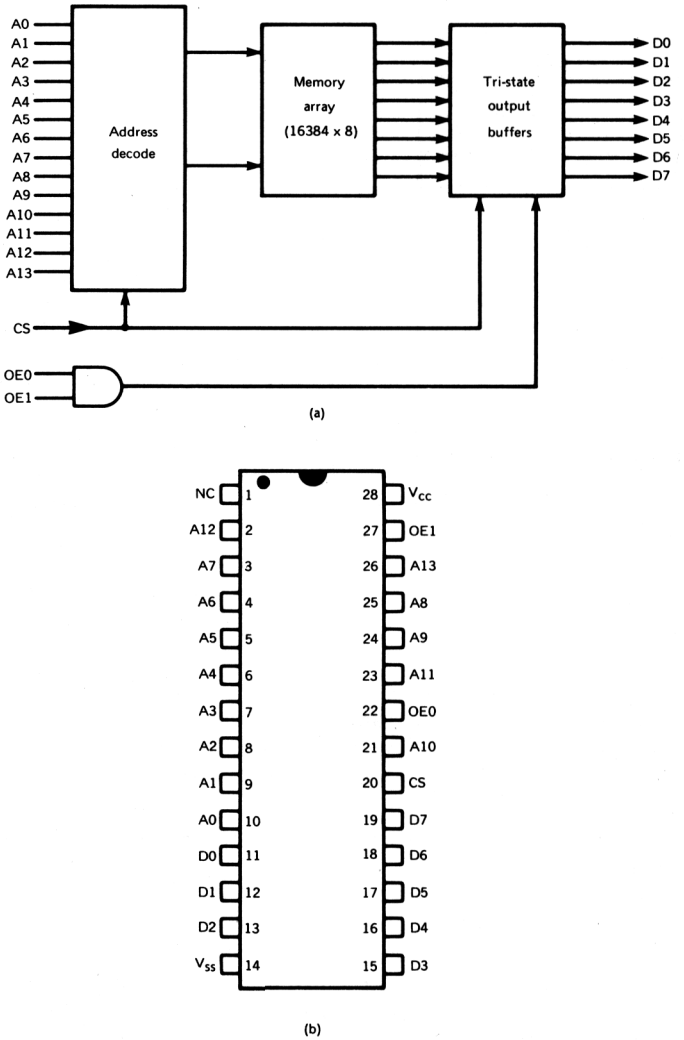


Fig. 9.9. (a) Block diagram of 16K HN613128P ROM. (b) ROM pin assignment

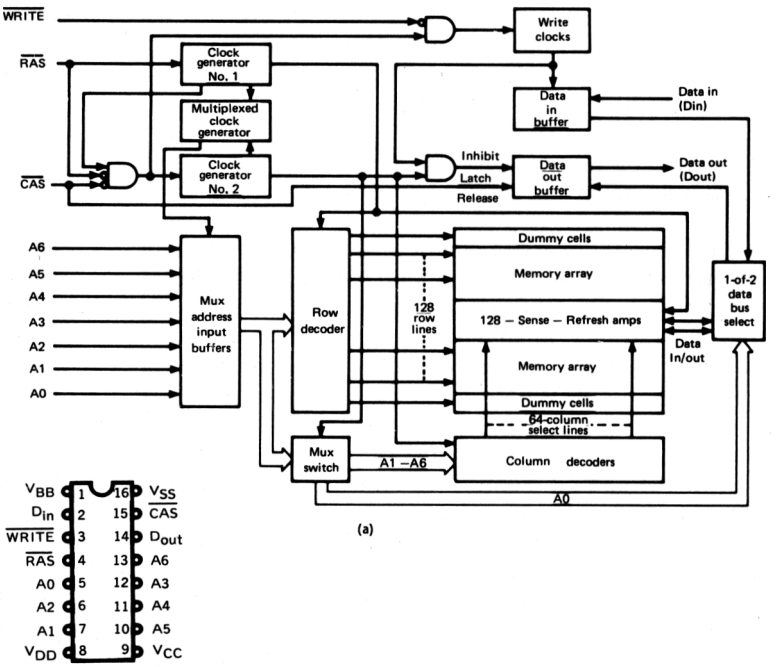
In Chapter 7 we described the form of the characters stored in ROM, and in Chapter 2 we explained how to examine the content of a memory location. You may recall that Program 4 in Appendix A may be used to dump the ROM contents.

Dynamic RAM

Dynamic random access memory (RAM) is used by the ZX Spectrum to store the program, data and system variables. It stores this information in the form of binary words (see Chapter 2), with each binary word held in an addressable memory location. This form of memory is *volatile*, that is, the data will only remain stored as long as the power supply is present. In the 16K Spectrum, to store a byte of information the computer uses eight 16384 × 1-bit dynamic RAM chips, type μPD416 or 4116 or equivalent. This implies that each of the 16 384 one-bit memory cells, which are arranged as a matrix of 128 rows by 128 columns, has a unique address; selection of a byte of memory from RAM is achieved by applying the address and appropriate control signals to all eight chips simultaneously.

Figure 9.10(a) shows the block diagram of a 16K dynamic RAM chip, and Fig. 9.10(b) gives the pin arrangement. The device has seven address inputs, A0 to A6, and two strobe signals, $\overline{RAS}$ and $\overline{CAS}$. At the start of a read/write memory cycle the microprocessor address bits, A0 to A6, are strobed and latched into the chip with $\overline{RAS}$ to address (select) one of the 128 rows of memory cells. This strobe signal also initiates the timing that enables the 128 column sense amplifiers. After a predetermined hold time the row address is removed and the microprocessor's seven high-order address bits, A7 to A13, are placed on the seven RAM address inputs, A0 to A6. This part of the full address is now strobed and latched into the device using the $\overline{CAS}$ strobe signal. Thus two of the 128 column sense amplifiers are selected by A1 to A6, and a one-of-two data bus select is achieved by A0. This completes the bit selection. Data may be then written into, or read from, the addressed memory cell.

The RAM chips used in the ZX Spectrum are dynamic, so the bit information in each cell is stored as a small quantity of electric charge; typically logic 0 is represented by zero charge and logic 1 by a charge of the order of 500 femto coulombs ($C \times 10^{-15}$). The memory cell storage capacitor loses its charge in time and therefore to preserve the stored bit the charge must be regenerated (refreshed) at least every two milliseconds. The Z80A provides an output signal $\overline{RFSH}$ and it has a refresh register R, which can be used as part of the refresh operation. Further details are given later in this chapter and in Chapter 10.



A0—A6	Address inputs
CAS	Column address strobe
D_{in}	Data in
D_{out}	Data out
RAS	Row address strobe
WRITE	Read/write input
V_{BB}	Power (—5V)
V_{CC}	Power (+5V)
V_{DD}	Power (+12V)
V_{SS}	Ground

(b)

Fig. 9.10. (a) Block diagram of 16K × one-bit dynamic RAM. (b) dynamic RAM pin assignment

In the 48K Spectrum, 16K of memory is supplied by eight 16K×1-bit dynamic RAMs of the type described above, and the additional 32K of memory is supplied by eight 32K×1-bit dynamic RAM chips, type TMS4532 or equivalent. The additional 32K of memory operates in a similar manner to the 16K, but is address decoded in the address range 32768 to 65535.

You may, of course, examine the content of any addressable RAM location using the **PEEK** function, or write into a specified address using the **POKE** statement as described in Chapter 2. You may recall that we used the **PEEK** and **POKE** functions in Chapter 7 under the headings User defined graphics, Character-grid attributes and More about the screen. **PEEK**ing and **POKE**ing the one-byte memory-mapped interface is discussed later in this chapter.

The memory map

When the Z80A microprocessor is addressing memory it will be either accessing a RAM location or a ROM location using the appropriate memory address, which is placed on the address bus. Clearly then, each memory location, whether in RAM or in ROM, has a unique address defined by the address bus and chip select lines.

The microcomputer system shown in Fig. 9.1 uses the address decoding logic within the Ferranti ULA chip, and this enables memory access to any one of the 16K locations in the ROM or any of the 16K locations in the RAM (or 48K if you have the 48K machine). The corresponding ROM and RAM address ranges are:

ROM 0000 to 16383

RAM { 16384 to 32767 (16K RAM)
RAM { 16384 to 65535 (48K RAM)

The specified memory ranges are often shown diagrammatically in the form known as a memory map, which illustrates the specified memory ranges for each system device or peripheral. Figure 9.11 shows the ZX Spectrum ROM and RAM memory-mapped addresses.

The Z80A microprocessor input and output signals

To enable you to connect peripheral hardware to your ZX Spectrum (printer, microdrive, I/O devices, etc) the MPU pin connections are brought out to the exposed edge connector (excepting ϕ, see below). Only when you understand the function of each MPU signal will you be in a position to design and connect your own interface circuits. To assist we give the pin assignment of the Z80A in Fig. 9.12 and summarise the functions of the signals.

The 40-pin dual-in-line NMOS Z80A is operated from a single +5V power supply.

The externally generated single-phase TTL-level clock, ϕ, drives the MPU control and timing logic, and defines the T-cycle period (see Chapter 10). The ϕ waveform does not appear on the edge counter, but the pin CK on the connector contains a signal of the same frequency as ϕ (3.5 MHz), but it has only a 1.5V amplitude with respect to the 0V line (see Fig. 10.3).

The tri-state, active high, eight-bit input/output bidirectional data lines, D0 (least significant bit) to D7 (most significant bit), are used for data transfers between the MPU and memory and I/O devices. The tri-state, active high, 16-bit output unidirectional address lines, A0 (least significant bit) to A15 (most significant bit) are used for setting up the address for memory and I/O device data transfers.

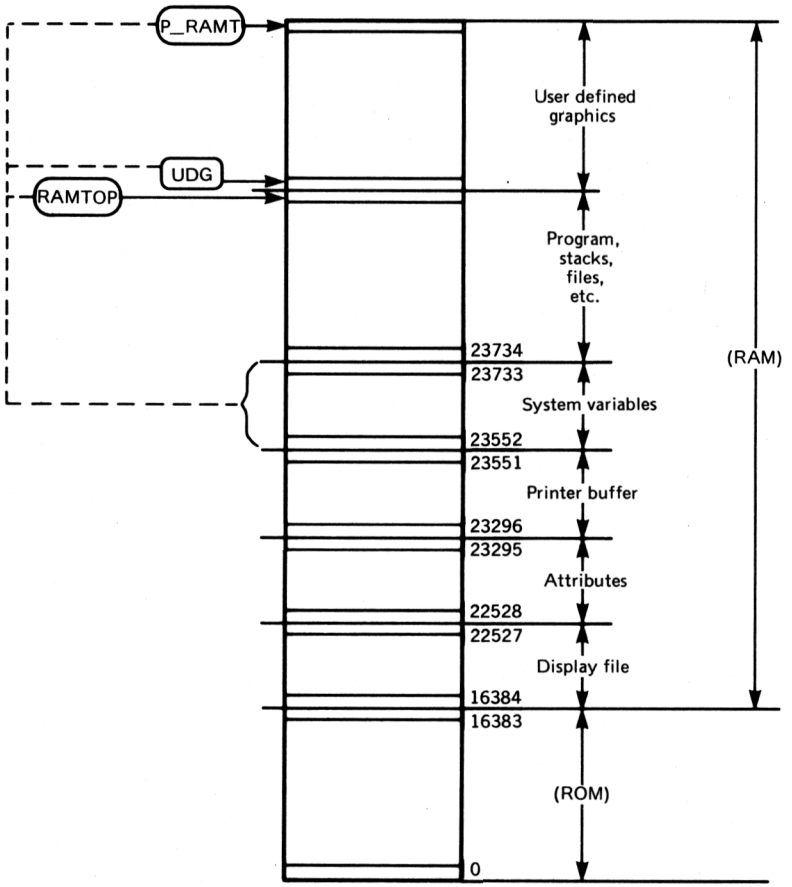


Fig. 9.11. Spectrum memory map

The control bus consists of five input and eight output connections. The five inputs are:

1. $\overline{\text{WAIT}}$. This is an active low input which makes the MPU enter *wait states*. These are ineffective T-cycles inserted into a machine cycle and the MPU idles until the $\overline{\text{WAIT}}$ is set high. Therefore it is possible to control the MPU so that it will wait until memory or I/O devices are ready to make transfers along the data bus.
2. $\overline{\text{RESET}}$. This is an active low input which sets the Program Counter to zero, disables the interrupt enable flip-flop, sets interrupt operation to mode 0, and clears Registers R and I.
3. $\overline{\text{INT}}$. This is the active low interrupt request input which responds to a signal generated by an I/O device when the interrupt enable flip-

flop (IFF) is enabled, and when the BUSRQ signal is inactive. The response to an accepted INT input signal is determined by the specified interrupt mode code (see Fig. 10.6).

4. NMI. This is the active low, negative-edge triggered, non-maskable interrupt input. It has higher priority than INT and is implemented at the end of the current instruction irrespective of the state of the interrupt enable flip-flop (IFF). The response to an acceptable NMI request is to make the MPU vector to memory location 102 (ie hexadecimal 0066). The NMI signal will not be accepted if WAIT or BUSRQ are active.
5. BUSRQ. This is the active low bus request input, and when enabled it sets the MPU address and data bus signals and the tri-state control output signals to the high impedance state. It is used in cases when external devices control the buses. The high impedance state is activated at the end of the machine cycle in which the BUSRQ signal is activated.

The eight output connections are:

1. BUSAK. This is an active low bus acknowledge output which, when active, indicates that the MPU address and data bus and tri-state control bus signals are in the high impedance state.
2. HALT. After executing a software Halt instruction, this output goes active low, and the MPU continues executing NOPs (the instruction for no operation). It remains in this state until an interrupt signal (NMI or INT) is received.
3. RFSH. This is an active low output which can be used with dynamic memories to refresh stored data. When the address bus lines A0 to A6 contain a refresh address this output is active.
4. WR. When the data bus holds valid data this tri-state active low output indicates this condition to an addressed device and the data may be then written into the device.
5. RD. When this tri-state output is active low, it may be used by an addressed device to recognise that the MPU is ready to read data from the data bus.
6. IORQ. This active low tri-state output indicates that a valid memory address is present on address lines A0 to A7.
7. MREQ. This active low tri-state output indicates that a valid memory address is present on address lines A0 to A15.
8. M1. This output will be active low during machine cycle M1 (see Chapter 10).

In the following section we show how some of the Z80A signals are used when implementing a one-byte memory-mapped interface and in Chapter 10 we consider machine-code programming of the Z80A microprocessor.

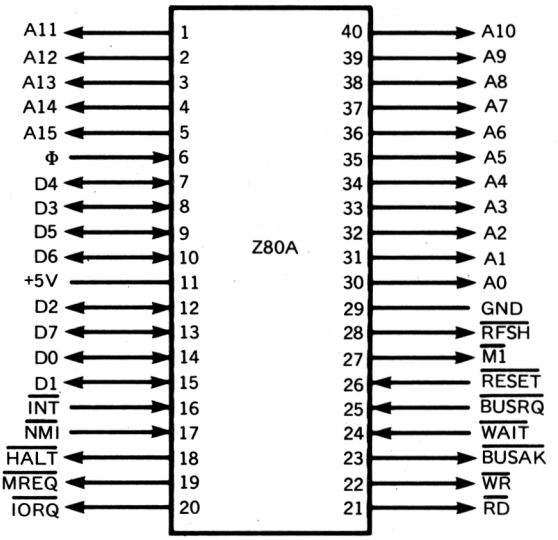


Fig. 9.12. Pin assignment of the Z80A microprocessor

One-byte memory-mapped interface

The ZX Spectrum address, data and control bus connections are provided at the exposed edge connector (see Fig. 9.13 and Table 9.1).

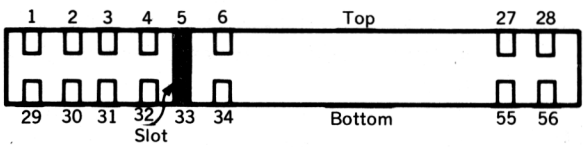


Fig. 9.13. Pin number assignment for edge connector

To input and output data from or to your external hardware, for example using **PEEK** and **POKE**, appropriate address decoding logic must be provided to ensure that your hardware is memory-mapped. Furthermore, you have to include control signal logic for determining whether a ZX Spectrum read or write operation is to be implemented (see Fig. 9.14). Your source of data must provide acceptable binary signals to the data bus for the ZX Spectrum memory read cycle. However, when the data bus is required by another device the input source connections must be isolated from the data bus. This can be done by using tri-state data bus buffers. In contrast, when the ZX

Table 9.1. Connections for the 56-way edge connector

Pin number	*Signal*	*Pin Number*	*Signal*
1	A15	29	A14
2	A13	30	A12
3	D7	31	+5V
4	not used	32	+9V
5	SLOT	33	SLOT
6	D0	34	0V
7	D1	35	0V
8	D2	36	CK
9	D6	37	A0
10	D5	38	A1
11	D3	39	A2
12	D4	40	A3
13	$\overline{\text{INT}}$	41	IORQGE
14	$\overline{\text{NMI}}$	42	0V
15	$\overline{\text{HALT}}$	43	VIDEO
16	$\overline{\text{MREQ}}$	44	Y
17	$\overline{\text{IORQ}}$	45	V
18	$\overline{\text{RD}}$	46	U
19	$\overline{\text{WR}}$	47	BUSRQ
20	−5V	48	RESET
21	$\overline{\text{WAIT}}$	49	A7
22	+12V	50	A6
23	−12V	51	A5
24	$\overline{\text{M1}}$	52	A4
25	$\overline{\text{RFSH}}$	53	$\overline{\text{ROMCS}}$
26	A8	54	$\overline{\text{BUSACK}}$
27	A10	55	A9
28	not used	56	A11

Spectrum writes data to the bus the device being written to must be capable of capturing the data during the memory write cycle. An octal latch is therefore suitable for this purpose.

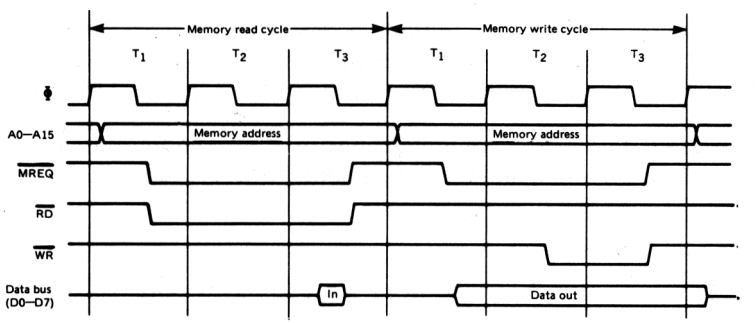


Fig. 9.14. MPU read or write cycles

Figure 9.15 shows the logic diagram of a one-byte memory-mapped interface. The address decoding logic maps this input/output port to the unique RAM address 32767 (7FFF), which is the last byte in the user defined graphics area of RAM in the 16K Spectrum. During the memory read cycle the octal data bus buffers (74LS244) are enabled by the $\overline{RD}$ signal, thereby connecting the external inputs onto the data bus. During the memory write cycle the octal latch (74LS373) is enabled by the $\overline{WR}$ signal and the data is latched and displayed on the light-emitting diode (LED) array. For reference purposes the pin assignments for the integrated circuits used in the one-byte interface of Fig. 9.15 are given in Fig. 9.16(a) to 9.16(f).

To read and display one byte of information derived from this external hardware to the ZX Spectrum use

PRINT PEEK 32767

You can use the simple program below to test the input port

```
3 LET x = PEEK 32767
5 PRINT AT 1Ø,14;x
7 PAUSE 2Ø
8 PRINT AT 1Ø,14;"   ": GO TO 3
```

This program periodically reads in the data byte provided by the user hardware and displays the decimal equivalent of the eight-bit number on the screen.

To write one data byte to the external hardware use

POKE 32767, data byte

The simple program below can be used to test the output port

```
 5 FOR n = 1 TO 255
 6 PAUSE 2Ø
1Ø POKE 32767,n
3Ø NEXT n
```

This program periodically outputs a data byte to the LED array, starting with the binary number equal to 1, and produces a binary counting operation, which continues until all of the LEDs are illuminated.

An alternative way of communicating with an input/output port is to use the **IN** function and **OUT** statement. The **IN** function takes the form

IN port address

and this reads the data byte from the addressed port. The **OUT** statement takes the form

OUT port address, data

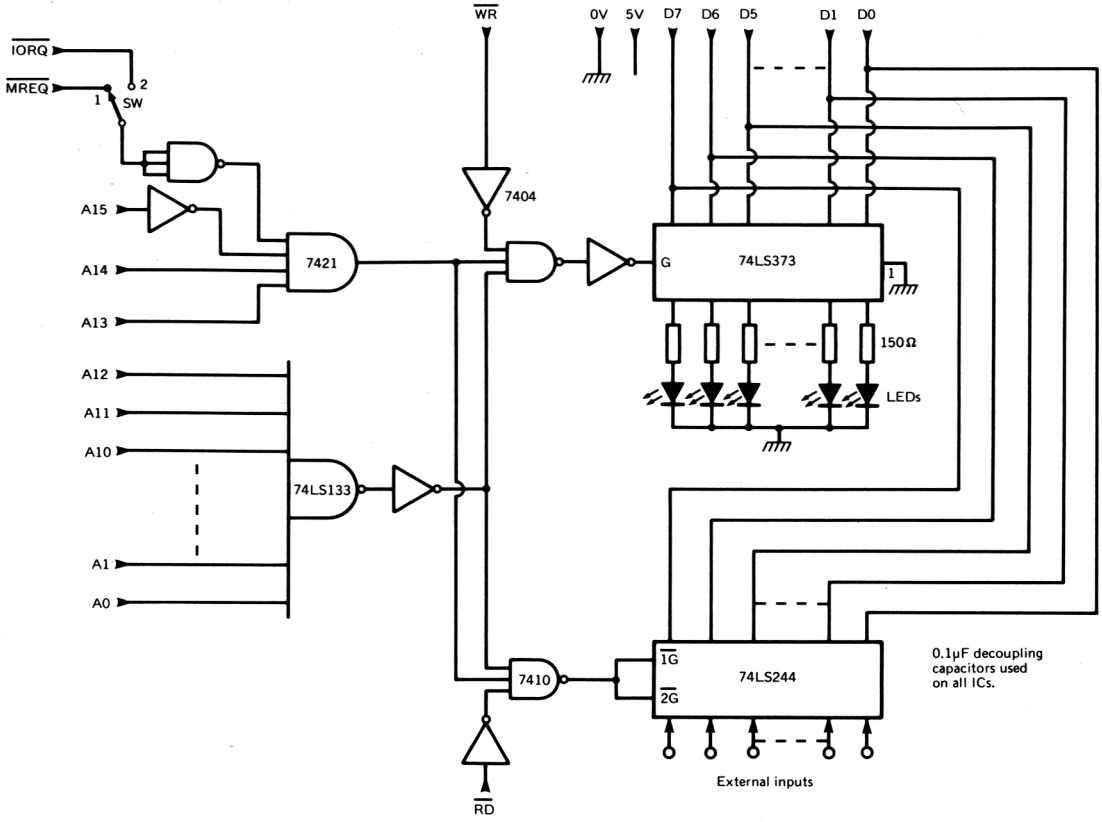
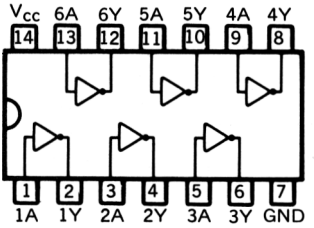
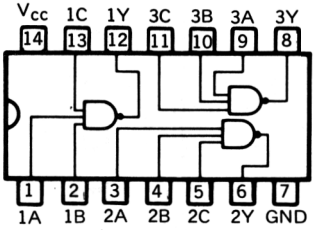


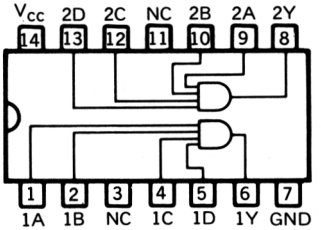
Fig. 9.15. One-byte memory-mapped interface (all the connecting leads between the interface logic and ZX Spectrum to be as short as possible)



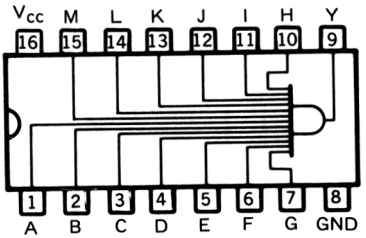
(a)



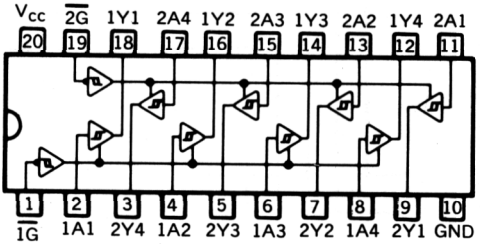
(b)



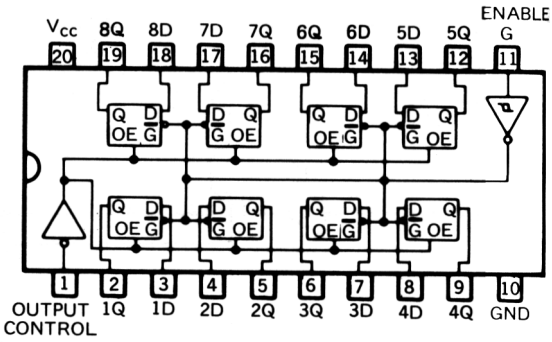
(c)



(d)



(e)



(f)

Fig. 9.16. Pin number assignments for one-byte memory-mapped interface. (a) 7404 (hex inverter), (b) 7410 (triple three-input NAND gate), (c) 7421 (dual four-input AND gate), (d) 74133 (13-input NAND gate), (e) 74LS244 (octal three-state driver), (f) 74LS373 (octal D-type transparent latch)

and this outputs the data to the addressed port.

These may appear to be the same as **PEEK** and **POKE**, but **IN** and **OUT** do not change the content of the memory location having the same address as the I/O port. You can test the output port using

```
 5 FOR n = 1 TO 255
 6 PAUSE 2Ø
1Ø OUT 32767,n
3Ø NEXT n
```

Consequently you do not need to reserve memory addresses for I/O ports.

In Chapter 10 the one-byte memory-mapped I/O port is used with machine code versions of **IN** and **OUT.**

To use the one-byte memory-mapped interface shown in Fig. 9.15 with **IN** and **OUT**, you should ensure that the switch, SW, is in position 2, thereby selecting control signal $\overline{IORQ}$ to implement the required timing (see Fig. 9.17).

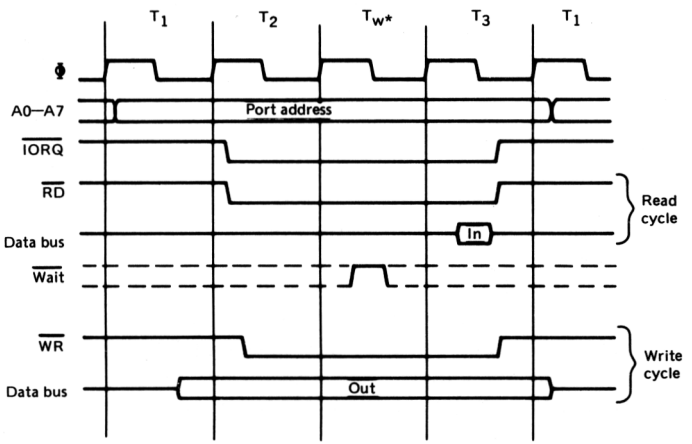


Fig. 9.17. MPU input/output cycles (*wait period inserted by Z80A)

The simple program below can be used to test the input port

```
3 LET x = IN 32767
5 PRINT AT 1Ø,14;x
7 PAUSE 2Ø
8 PRINT AT 1Ø,14;"  " GO TO 3
```

Concluding remarks

In this chapter we have given an insight into some of the important aspects of the computer hardware, and the reference material provided will be useful when you undertake the design and implementation of peripheral hardware.

10

Programming in machine code

Introduction

When we write programs in Basic our statements, functions and numbers have to be stored and subsequently translated to equivalent machine code by the ZX Spectrum interpreter and operating system. The Basic commands are interpreted so that the Z80A microprocessor can fetch and execute the instructions and this is necessary because the Z80A can work only with machine code.

The interpretation process takes a finite time to implement, and if this can be eliminated by writing programs in machine code rather than Basic, then a significant time saving is achieved. This can be an important consideration in applications involving graphics or control of external peripheral devices.

The number of memory bytes required for a program written in Basic is much larger than the number required for the equivalent machine code program, so a significant saving in memory requirements is achieved by writing programs in machine code.

Let us now consider how the Z80A microprocessor implements instructions, and examine its instruction set and addressing modes. We will also see how machine code programs can be run on the ZX Spectrum.

Implementation of instructions by the Z80A microprocessor

In Fig. 10.1 there is a functional block diagram of the Z80A, and you will note that there are sixteen address lines, eight data lines and thirteen control lines. You may recall that in Chapter 9 we described the pin assignment and signals of the Z80A.

The Z80A (MPU) addresses the system devices (RAM, ROM and

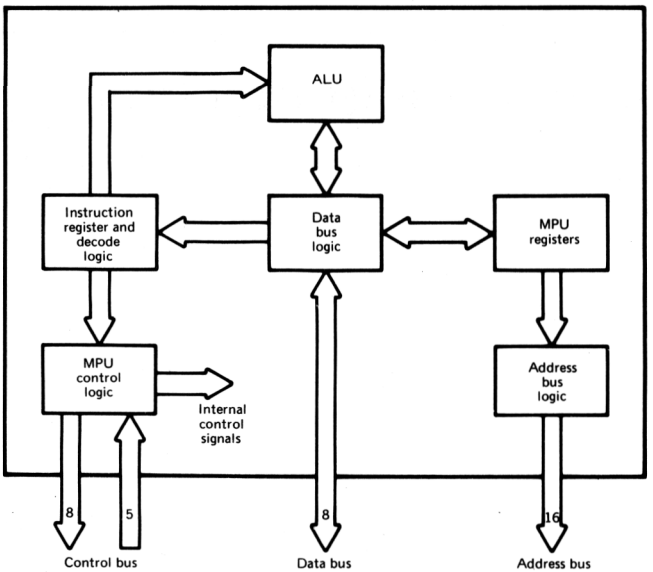


Fig. 10.1. Functional block diagram of the Z80A

I/O) via the 16-bit address bus, and is thus capable of addressing 65536(2^{16}) unique memory locations in the range 0000 to FFFF. The 16 address lines may be conveniently partitioned into two bytes: address bits A0-A7 form the least significant byte (LS byte) of the address, and address bits A8-A15 form the most significant byte (MS byte) of the address. For example:

$$\underbrace{11000000}_{\text{MS byte}}\ \underbrace{10011110}_{\text{LS byte}}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{16-bit address bus} \equiv \text{C09E}}$$

The eight-bit bidirectional data bus of the MPU is used to transmit data to, or receive data from, the RAM, ROM and I/O devices. A data bus word will, for example, have the form

$$\underbrace{10001111}_{\text{8-bit data bus} \equiv \text{8F}}$$

The 16-bit address held in the MPU Program Counter register points to the next stored operation code (op-code) to be transferred to the MPU Instruction Register on receipt of the appropriate timing

and control signals. The content of the Instruction Register is then decoded and the appropriate fetch, store, arithmetic or logic operation is executed by the MPU.

To illustrate the principle of the MPU *fetch and execute* operation consider the instruction which immediately loads Register D in the MPU with data of value 6E. From the Instruction Set summary given in Table 10.1 we see that the op-code for loading Register D using Immediate Addressing (LD r,n) is 00010110 (hexadecimal 16) which we will assume is stored in RAM location 27000 (hexadecimal 6978). The data word 6E will be stored in the RAM location immediately following the op-code, that is, at address 27001 (hexadecimal 6979).

Figure 10.2 shows the timing diagram for this example and you may note that the complete instruction cycle consists of two machine cycles. The first machine cycle, M1, is used to fetch the op-code from memory for instruction decoding, and the second machine cycle, M2, is used to read the data byte into Register D. After completing this instruction the Program Counter will have been incremented to point to the next stored op-code, that is, the Program Counter will point to memory location 27002 (hexadecimal 697A).

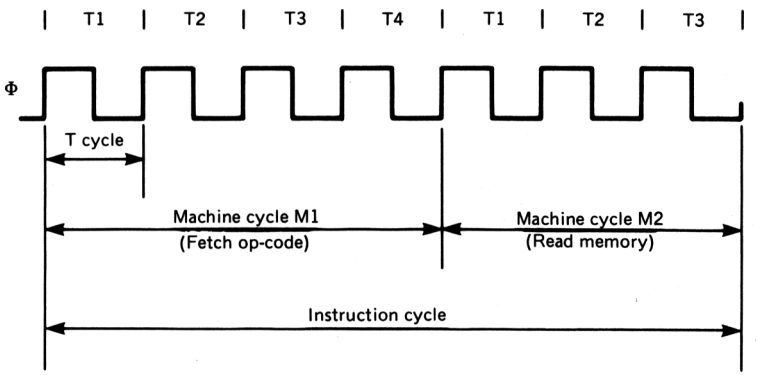


Fig. 10.2. Timing diagram for Immediate Addressing example

The two M cycles are of different duration: M1 contains four T-cycles, while M2 contains three T-cycles, where the duration of a T-cycle corresponds to the MPU ϕ clock period. Obviously the number of T-cycles per instruction cycle determines the time to fetch and execute an instruction, and this depends upon the complexity of the instruction.

The waveform shown in Fig. 10.3 is an oscillogram of the clock used in the Spectrum which is provided on pin CK of the exposed edge connector. The measured value of the T-cycle period is 285 nanoseconds (285×10^{-9}s). Therefore in the above example

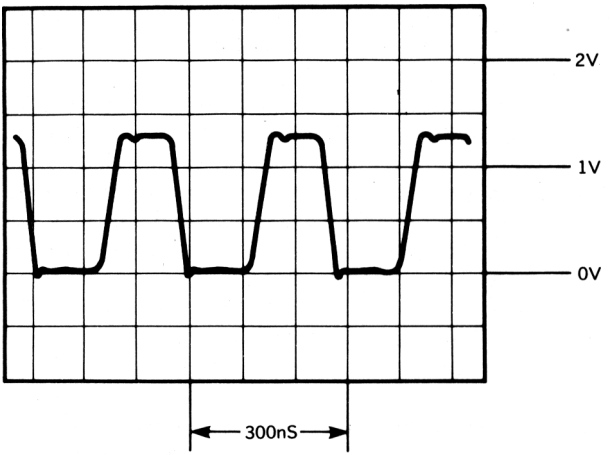


Fig. 10.3. Edge connector CK waveform

involving seven T-cycles the time to fetch and execute the instruction is 7×285 nanoseconds = 1.995 microseconds (1.995×10^{-6}s). The above T-cycle period corresponds to the Z80A microprocessor running at 3.5 MHz.

Z80A accessible registers

The MPU register configuration for the Z80A microprocessor is shown in Fig. 10.4 and it can be seen that it contains 22 program-accessible internal registers.

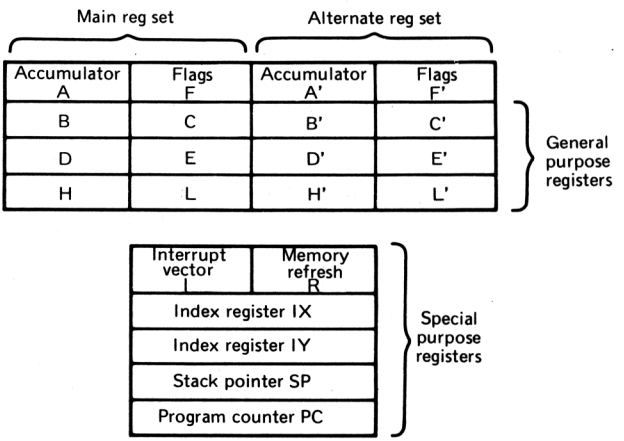


Fig. 10.4. Z80A program-accessible registers

There are two sets of general purpose registers, each set containing six 8-bit registers, and two sets of accumulator and flag registers. The general purpose eight bit registers in each set may be concatenated (paired) to form 16-bit register pairs BC, DE and HL for the main register set, and B'C', D'E' and H'L' for the alternate register set.

The programmer, by using a single exchange instruction, can work with either the main register set or the alternate register set. This feature is useful where fast response to an interrupt is required. In such cases the interrupt service routine can be programmed to substitute quickly the alternate register set for the main register set, and then switch back at the end of the interrupt routine.

The accumulator and flag register in the main register set or alternate register set are selected with a single exchange instruction. The eight-bit accumulator holds the result of arithmetic or logical operations and the flag register holds the states corresponding to the occurrence of specific events resulting from the execution of instructions (see Fig. 10.5). For example, the carry flag C contains the highest order bit of the accumulator according to the instruction executed.

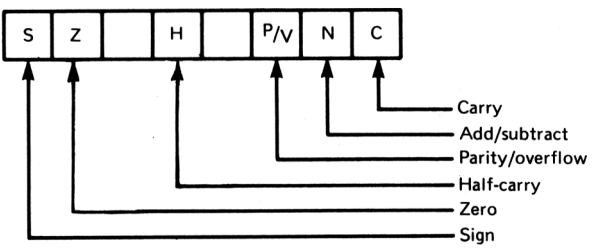


Fig. 10.5. Flag register format

The Z80A also contains four 16-bit registers, IX, IY, SP and PC, an eight-bit I register, and a seven-bit R register. These are special-purpose registers and are described below.

The two 16-bit registers IX and IY are independent index registers that are used to hold 16-bit base addresses. These are used with indexed addressing mode instructions, whereby a two's complement signed integer (included as an additional byte in the instruction) specifies the displacement from the base address and this yields the address of the memory location to be accessed.

The 16-bit Stack Pointer, SP, contains the address of the current top of the stack, which is RAM that accepts or outputs data in a last-in, first-out (LIFO) mode of operation. That is, data can be pushed onto the stack from specified MPU registers or popped off the stack to specified MPU registers.

The 16-bit program Counter, PC, contains the memory address of the instruction currently being fetched from memory. Its content is changed in accordance with the requirements to fetch and execute the next instruction.

The eight-bit I register is the Interrupt Page Address Register. It is used when an interrupt is sensed by the MPU, and it stores the high order eight bits of the vector table address. The interrupting device supplies the lower eight bits of the vector table address (see Fig. 10.6).

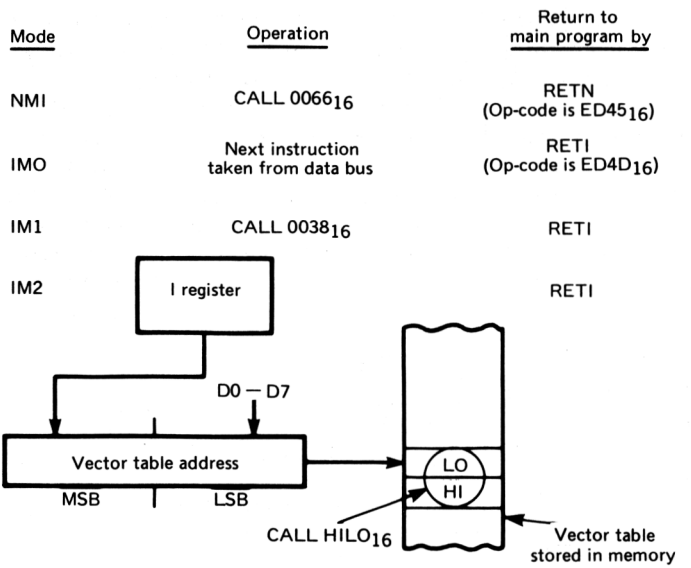


Fig. 10.6. Interrupt mode summary

The seven-bit Memory Refresh Register, R, is a counter that is automatically incremented after each M1 cycle (see Fig. 10.2). It contains the addresses A0 to A6 which may be used to access dynamic memory locations, and which are then refreshed by the $\overline{\text{RFSH}}$ Z80A output signal. This refresh operation does not interfere with normal MPU operation and it is transparent to the programmer.

Z80A Instruction Set summary

It is useful to the user to have the 158 different types of Z80A instructions logically arranged in groups as shown in Tables 10.1 to 10.11. For the instructions in each table the assembly language mnemonic, the symbolic operation, the status of the Flag Register bits, the op-code (binary or hex form) and relevant storage and timing

information is provided. Comments, notes and flag notation are also included.

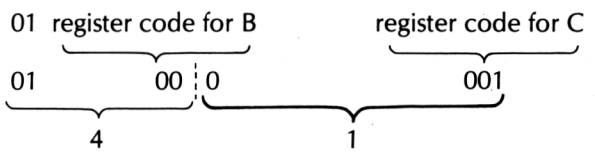
We shall see in later sections of this chapter how to use some of these instructions when writing machine code programs.

Addressing modes

We have seen how a machine code instruction is fetched and executed by the Z80A microprocessor. Furthermore, we have seen that there are 22 program-accessible registers in the Z80A, and we have examined its instruction set. However, to use this information effectively in writing machine code programs, you need to understand the various Z80A addressing modes, which are now described.

Register addressing

This mode of addressing has a single-byte op-code which defines the MPU registers involved. In Table 10.1 the first instruction, LD r,s, uses this form of addressing. For example, if we wish to copy (load) the content of register C into register B the op-code is



Thus the op-code 41, stored in a single byte of memory, implements the required operation. Note that as each register of the Z80A has a different three-bit code, you can copy the content of any register into another register using the appropriate op-code.

Register indirect addressing

The content of a general purpose register pair or a 16-bit special-purpose register forms the memory address of the data byte to be accessed. For example, to load register A with the accessed data in the memory address specified by the content of the DE register pair the instruction LD A, (DE) is used, ie from Table 10.1 it is seen that op-code 1A implements the instruction.

In contrast, as a further example of this form of addressing, consider the instruction POP IX. This loads the low byte of the index

Table 10.1. Z80A eight-bit load instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD r, s	r ← s	•	•	X	•	X	•	•	•	01 r s		1	1	4	r, s Reg.
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110 ← n →		2	2	7	000 B 001 C
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	010 D
LD r, (IX+d)	r ← (IX+d)	•	•	X	•	X	•	•	•	11 011 101 01 r 110 ← d →	DD	3	5	19	011 E 100 H 101 L
LD r, (IY+d)	r ← (IY+d)	•	•	X	•	X	•	•	•	11 111 101 01 r 110 ← d →	FD	3	5	19	111 A
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7	
LD (IX+d), r	(IX+d) ← r	•	•	X	•	X	•	•	•	11 011 101 01 110 r ← d →	DD	3	5	19	
LD (IY+d), r	(IY+d) ← r	•	•	X	•	X	•	•	•	11 111 101 01 110 r ← d →	FD	3	5	19	
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110 ← n →	36	2	3	10	
LD (IX+d), n	(IX+d) ← n	•	•	X	•	X	•	•	•	11 011 101 00 110 110 ← d → ← n →	DD 36	4	5	19	

LD (IY+d), n	(IY+d)←n	•	•	X	•	X	•	•	•	11 111 101 00 110 110 ← d → ← n →	FD 36	4	5	19
LD A, (BC)	A←(BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7
LD A, (DE)	A←(DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7
LD A, (nn)	A←(nn)	•	•	X	•	X	•	•	•	00 111 010 ← n → ← n →	3A	3	4	13
LD (BC), A	(BC)←A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7
LD (DE), A	(DE)←A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7
LD (nn), A	(nn)←A	•	•	X	•	X	•	•	•	00 110 010 ← n → ← n →	32	3	4	13
LD A, I	A←I	↕	↕	X	0	X	IFF	0	•	11 101 101 01 010 111	ED 57	2	2	9
LD A, R	A←R	↕	↕	X	0	X	IFF	0	•	11 101 101 01 011 111	ED 5F	2	2	9
LD I, A	I←A	•	•	X	•	X	•	•	•	11 101 101 01 000 111	ED 47	2	2	9
LD R, A	R←A	•	•	X	•	X	•	•	•	11 101 101 01 001 111	ED 4F	2	2	9

Notes: r, s means any of the registers A, B, C, D, E, H, L
IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10.2. Z80A 16-bit load instructions.

Mnemonic	Symbolic Operation	Flags								Op-Code			No. of Bytes	No. of M Cycles	No. of T States	Comments
		S	Z		H		P/V	N	C	76 543 210	Hex					
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →			3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21		4	4	14	
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21		4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	X	•	X	•	•	•	00 101 010 ← n → ← n →	2A		3	5	16	
LD dd, (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	X	•	X	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED		4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	X	•	X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A		4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	X	•	X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A		4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	X	•	X	•	•	•	00 100 010 ← n → ← n →	22		3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	X	•	X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED		4	6	20	

LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	X	•	X	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20		
LD (nn), IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	X	•	X	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20		
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6		
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101 11 111 001	DD F9	2	2	10		
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101 11 111 001	FD F9	2	2	10		
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11		
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H	•	•	X	•	X	•	•	•	11 011 101 11 100 101	DD E5	2	4	15		
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H	•	•	X	•	X	•	•	•	11 111 101 11 100 101	FD E5	2	4	15		
POP qq	qq_H ← (SP+1) qq_L ← (SP)	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10		
POP IX	IX_H ← (SP+1) IX_L ← (SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 001	DD E1	2	4	14		
POP IY	IY_H ← (SP+1) IY_L ← (SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 001	FD E1	2	4	14		

qq	Pair
00	BC
01	DE
10	HL
11	AF

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

Table 10.3. Z80A exchange group and block transfer and search instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE↔HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF'	AF↔AF'	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	(BC↔BC' DE↔DE' HL↔HL')	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H↔(SP+1) L↔(SP)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	IX_H↔(SP+1) IX_L↔(SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	IY_H↔(SP+1) IY_L↔(SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	
LDI	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	X	0	X	① ↕	0	•	11 101 101 10 100 000	ED A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	X	0	X	① ↕	0	•	11 101 101 10 101 000	ED A8	2	4	16	

LDDR	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	X	0	X	0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A – (HL) HL ← HL+1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A – (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A – (HL) HL ← HL-1 BC ← BC-1	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A – (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	↕	② ↕	X	↕	X	① ↕	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10.4. Z80A eight-bit arithmetic and logical instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	X	↕	X	V	0	↕	10 [000] r		1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	X	↕	X	V	0	↕	11 [000] 110		2	2	7	000 B
										← n →					001 C
															010 D
ADD A, (HL)	A ← A+(HL)	↕	↕	X	↕	X	V	0	↕	10 [000] 110		1	2	7	011 E
ADD A, (IX+d)	A←A+(IX+d)	↕	↕	X	↕	X	V	0	↕	11 011 101	DD	3	5	19	100 H
										10 [000] 110					101 L
										← d →					111 A
ADD A, (IY+d)	A←A+(IY+d)	↕	↕	X	↕	X	V	0	↕	11 111 101	FD	3	5	19	
										10 [000] 110					
										← d →					
ADC A, s	A←A+s+CY	↕	↕	X	↕	X	V	0	↕	[001]					s is any of r, n,
SUB s	A←A - s	↕	↕	X	↕	X	V	1	↕	[010]					(HL), (IX+d),
SBC A, s	A←A - s - CY	↕	↕	X	↕	X	V	1	↕	[011]					(IY+d) as shown for
AND s	A←A ∧ s	↕	↕	X	1	X	P	0	0	[100]					ADD instruction.
OR s	A←A ∨ s	↕	↕	X	0	X	P	0	0	[110]					The indicated bits
XOR s	A←A ⊕ s	↕	↕	X	0	X	P	0	0	[101]					replace the [000] in
CP s	A - s	↕	↕	X	↕	X	V	1	↕	[111]					the ADD set above.
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r [100]		1	1	4	

INC (HL)	(HL)←(HL)+1	↕	↕	X	↕	X	V	0	•	00 110 [100]		1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	↕	↕	X	↕	X	V	0	•	11 011 101 00 110 [100] ← d ←	DD	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d)+1	↕	↕	X	↕	X	V	0	•	11 111 101 00 110 [100] ← d ←	FD	3	6	23	
DEC s	s ← s - 1	↕	↕	X	↕	X	V	1	•	[101]					s is any of r, (HL), (IX+d), (IY+d) as shown for INC. DEC same format and states as INC. Replace [100] with [101] in OP Code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow, P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

Table 10.5. Z80A general-purpose arithmetic and MPU control instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD HL, ss	HL ← HL+ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11	ss Reg. 00 BC
ADC HL, ss	HL ← HL+ss+CY	↕	↕	X	X	X	V	0	↕	11 101 101 01 ss1 010	ED	2	4	15	01 DE 10 HL 11 SP
SBC HL, ss	HL ← HL-ss-CY	↕	↕	X	X	X	V	1	↕	11 101 101 01 ss0 010	ED	2	4	15	
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101 00 pp1 001	DD	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY ← IY + rr.	•	•	X	X	X	•	0	↕	11 111 101 00 rr1 001	FD	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6	
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101 00 100 011	DD 23	2	2	10	
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101 00 100 011	FD 23	2	2	10	
DEC ss	ss ← ss - 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6	
DEC IX	IX ← IX - 1	•	•	X	•	X	•	•	•	11 011 101 00 101 011	DD 2B	2	2	10	
DEC IY	IY ← IY - 1	•	•	X	•	X	•	•	•	11 111 101 00 101 011	FD 2B	2	2	10	

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

Table 10.6. Z80A 16-bit arithmetic instructions

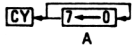
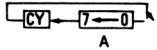
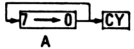
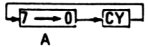
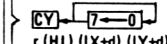
Mnemonic	Symbolic Operation	Flags								Op-Code		No. of Bytes	No. of M Cycles	No. of T States	Comments
		S	Z		H		P/V	N	C	76 543 210	Hex				
DAA	Converts acc, content into packed BCD following add or subtract with packed BCD operands	↕	↕	X	↕	X	P	•	↕	00 100 111	27	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	X	1	X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (One's complement)
NEG	$A \leftarrow \overline{A}+1$	↕	↕	X	↕	X	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Negate acc, (two's complement)
CCF	$CY \leftarrow \overline{CY}$	•	•	X	X	X	•	0	↕	00 111 111	3F	1	1	4	Complement carry flag
SCF	CY ← 1	•	•	X	0	X	•	0	1	00 110 111	37	1	1	4	Set carry flag
NOP	No operation	•	•	X	•	X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	•	•	01 110 110	76	1	1	4	
DI*	IFF ← 0	•	•	X	•	X	•	•	•	11 110 011	F3	1	1	4	
EI*	IFF ← 1	•	•	X	•	X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•	X	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•	X	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•	X	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

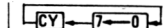
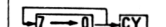
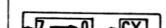
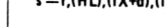
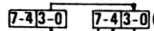
Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

*Interrupts are not sampled at the end of EI or DI

Table 10.7. Z80A rotate and shift instructions

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA		•	•	X	0	X	•	0	‡	00 000 111	07	1	1	4	Rotate left circular accumulator
RLA		•	•	X	0	X	•	0	‡	00 010 111	17	1	1	4	Rotate left accumulator
RRCA		•	•	X	0	X	•	0	‡	00 001 111	0F	1	1	4	Rotate right circular accumulator
RRA		•	•	X	0	X	•	0	‡	00 011 111	1F	1	1	4	Rotate right accumulator
RLC r	 r,(HL),(IX+d),(IY+d)	‡	‡	X	0	X	P	0	‡	11 001 011 00 000 r	CB	2	2	8	Rotate left circular register r
RLC (HL)		‡	‡	X	0	X	P	0	‡	11 001 011 00 000 110	CB	2	4	15	r Reg. 000 B 001 C
RLC (IX+d)		‡	‡	X	0	X	P	0	‡	11 011 101 11 001 011 ← d → 00 000 110	DD CB	4	6	23	010 D 011 E 100 H 101 L 111 A
RLC (IY+d)		‡	‡	X	0	X	P	0	‡	11 111 101 11 001 011 ← d → 00 000 110	FD CB	4	6	23	

RL s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	010						Instruction format and states are as shown for RLC's. To form new Op-Code replace 000 of RLC's with shown code
RRC s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	001						
RR s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	011						
SLA s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	100						
SRA s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	101						
SRL s	 s ≡ r,(HL),(IX+d),(IY+d)	↕	↕	X	0	X	P	0	↕	111						
RLD	A  (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18		Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD	A (HL)	↕	↕	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18		

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10.8. Z80A bit set, reset and test instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No.of M Cycles	No.of T States	Comments
BIT b, r	$Z \leftarrow \overline{r_b}$	X	‡	X	1	X	X	0	•	11 001 011 01 b r	CB	2	2	8	
BIT b, (HL)	$Z \leftarrow \overline{(HL)_b}$	X	‡	X	1	X	X	0	•	11 001 011 01 b 110	CB	2	3	12	
BIT b, $(IX+d)_b$	$Z \leftarrow \overline{(IX+d)_b}$	X	‡	X	1	X	X	0	•	11 011 101 11 001 011 ← d → 01 b 110	DD CB	4	5	20	
BIT b, $(IY+d)_b$	$Z \leftarrow \overline{(IY+d)_b}$	X	‡	X	1	X	X	0	•	11 111 101 11 001 011 ← d → 01 b 110	FD CB	4	5	20	
SET b, r	$r_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011 11 b r	CB	2	2	8	

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

b	Bit Tested
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011 [11] b 110	CB	2	4	15	
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 011 101 11 001 011 ← d → [11] b 110	DD CB	4	6	23	
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 101 11 001 011 ← d → [11] b 110	FD CB	4	6	23	
RES b, s	$s_b \leftarrow 0$ $s \equiv r$, (HL), (IX+d), (IY+d)	•	•	X	•	X	•	•	•	[10]					To form new O_p. Code replace [11] of SET b, s with [10] Flags and time states for SET instruction

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10.9. Z80A jump instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	X	•	X	•	•	•	11 000 011 ← n → ← n →	C3	3	3	10	
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	X	•	X	•	•	•	11 cc 010 ← n → ← n →		3	3	10	cc | Condition 000 NZ non zero 001 Z zero 010 NC non carry 011 C carry 100 PO parity odd 101 PE parity even 110 P sign positive 111 M sign negative
JR e	PC ← PC + e	•	•	X	•	X	•	•	•	00 011 000 ← e-2 →	18	2	3	12	
JR C, e	If C = 0, continue	•	•	X	•	X	•	•	•	00 111 000 ← e-2 →	38	2	2	7	If condition not met
	If C = 1, PC ← PC+e											2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	X	•	X	•	•	•	00 110 000 ← e-2 →	30	2	2	7	If condition not met
	If C = 0, PC ← PC+e											2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	X	•	X	•	•	•	00 101 000 ← e-2 →	28	2	2	7	If condition not met
	If Z = 1, PC ← PC+e											2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	X	•	X	•	•	•	00 100 000 ← e-2 →	20	2	2	7	If condition not met
	If Z = 0, PC ← PC+e											2	3	12	If condition is met

JP (HL)	PC ← HL	•	•	X	•	X	•	•	•	11 101 001	E9	1		1	4	
JP (IX)	PC ← IX	•	•	X	•	X	•	•	•	11 011 101 11 101 001	DD E9	2		2	8	
JP (IY)	PC ← IY	•	•	X	•	X	•	•	•	11 111 101 11 101 001	FD E9	2		2	8	
DJNZ, e	B ← B-1 If B = 0, continue	•	•	X	•	X	•	•	•	00 010 000 ← e-2 ←	10	2		2	8	If B = 0
	If B ≠ 0, PC ← PC+e											2		3	13	If B ≠ 0

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <126, 129>

e-2 in the op-code provides an effective address of pc+e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10.10. Z80A call and return instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No. of Bytes	No.of M Cycles	No.of T States	Comments
CALL nn	(SP-1) ← PC_H (SP-2) ← PC_L PC ← nn	•	•	X	•	X	•	•	•	11 001 101 ← n ← ← n ←	CD	3	5	17	
CALL cc, nn	If condition cc is false	•	•	X	•	X	•	•	•	11 cc 100 ← n ←		3	3	10	If cc is false
	continue, otherwise same as CALL nn									← n ←		3	5	17	If cc is true
RET	PC_L ← (SP) PC_H ← (SP+1)	•	•	X	•	X	•	•	•	11 001 001	C9	1	3	10	
RET cc	If condition cc is false	•	•	X	•	X	•	•	•	11 cc 000		1	1	5	If cc is false
	continue, otherwise same as RET											1	3	11	If cc is true

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry

RETI	Return from interrupt	•	•	X	•	X	•	•	•	11 101 101 01 001 101	ED 4D	2	4	14	011	C	carry
RETN[1]	Return from non maskable interrupt	•	•	X	•	X	•	•	•	11 101 101 01 000 101	ED 45	2	4	14	100 101 110 111	PO PE P M	parity odd parity even sign positive sign negative
RST p	(SP-1) ← PC_H (SP-2) ← PC_L PC_H ← 0 PC_L ← p	•	•	X	•	X	•	•	•	11 t 111		1	3	11			

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

[1] RETN loads IFF_2 ← IFF_1

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table 10.11. Z80A input and output instructions

Mnemonic	Symbolic Operation	Flags S	Z		H		P/V	N	C	Op-Code 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011 ← n →	DB	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	↕	↕	X	↕	X	P	0	•	11 101 101 01 r 000	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	X	↕ ①	X	X	X	X	1	•	11 101 101 10 100 010	ED A2	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 010	ED B2	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X	↕ ①	X	X	X	X	1	•	11 101 101 10 101 010	ED AA	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 010	ED BA	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011	D3	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101 01 r 001	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 100 011	ED A3	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 110 011	ED B3	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	X	X	1	•	11 101 101 10 101 011	ED AB	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	X	X	1	•	11 101 101 10 111 011	ED BB	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

register IX with the content of the data held at the memory address given by the content of the Stack Pointer, SP, and loads the high byte of the index register with the data held at the memory address given by the Stack Pointer plus one. From Table 10.2 we can see that the op-code for the two-byte instruction is:

DD
E1

Immediate addressing

In this mode of addressing the data byte *immediately* follows the op-code. For example, the instruction of the form LD r,n loads register r with the byte n. Thus to preset register E with data equal to F8 we see from Table 10.1 that the instruction is

1E Op-code
F8 Data

Immediate extended addressing

This form of addressing can be used to preset register pairs and the 16-bit special-purpose registers. For example, using Table 10.2, we can see that the instruction

DD 21	Op-code
FF	Data LO
ØØ	Data HI

loads the Index Register IX with the data ØØFF, and the instruction

Ø1	Op-code
BA	Data LO
AB	Data HI

loads the register pair BC with the data ABBA.

Extended addressing

The last two bytes of the instruction are used to specify the address of the memory location for the data to be used with the instruction, or they contain an address used with a jump instruction. For example, the instruction LD A,(nn) loads accumulator A with the content of memory location having the address nn. From Table 10.1 we can see

that to load accumulator A with the content of memory location 1982 we use the instruction

3A	Op-code
82	Address LO
19	Address HI

A further example is the instruction of the form LD dd, (nn), which loads register pair dd (see the two-bit code in Table 10.2) with data from memory addresses nn and nn + 1, where the content of the memory byte addressed by nn is loaded into the LO register of the pair and the content of the memory byte addressed by nn + 1 is loaded into the HI register of the pair. Thus to load register pair BC with the data held in memory locations 1982 and 1983 the required instruction is

ED 4B	Op-code	
82	LO	Address of data for register C
19	HI	

A straightforward form of jump instruction that uses this form of addressing, is the Unconditional Jump, JP, nn (see Table 10.9). For example, the instruction

C3	Op-code
21	Address LO
C4	Address HI

will make the program jump to memory address C421, thereby accessing the first byte of the next instruction.

In contrast, the jump instruction can be made conditional upon the state of a specified flag condition by using the JP cc,nn instruction given in Table 10.9. For example, the instruction

CA	Op-code
2B	Address LO
14	Address HI

will cause the program to jump to memory location 142B *if* the result of the last instruction was zero. If not, the jump instruction is ignored and the next consecutive instruction in the program is executed.

Modified page zero addressing

The eight one-byte restart instructions, RST p, given in Table 10.10, use modified page zero addressing to direct program control to one of the eight predetermined addresses, which have a HI address byte

of ØØ, hence the name given to this addressing mode. The instruction to gain access to the predetermined memory location requires only the single-byte op-code, without the need for any additional address bytes. This results in a saving of memory space and time. This is a useful feature for accessing frequently used subroutines.

The ZX Spectrum ROM uses the eight restart instructions in its monitor routines. For example, the RST to memory address ØØØ8 is used for the start address of the Error Report Handling Routine. As shown in Table 10.10 the op-code CF is used to access ØØØ8. To use this facility in machine code programs to generate your own report codes use the op-code CF followed by the data byte required to define the alphanumeric report code character to be displayed. The data byte used for reports A to R is obtained by subtracting 38 (hex) from the hex code, as defined in Table 7.1. For example, to display report code P (hex code 5Ø), use the instruction

```
CF      Op-code
18      Data byte (5Ø-38)
```

The data byte used for reports Ø-9 is obtained by subtracting 31 (hex) from the hex code given in Table 7.1.

Implied Addressing

Instructions which operate with the content of a specific register are known as implied addressing mode instructions. For example, this is the case for arithmetic and logic operations on contents of accumulator A. This form of addressing is illustrated in the following examples:

1. To increment the content of a selected register we can use the instruction INC r. For instance, to increment the content of register B we can use the op-code Ø4, obtained from Table 10.4.
2. To form the one's complement of the content of accumulator A use the instruction CPL. The corresponding op-code is 2F, see Table 10.5.
3. To shift the content of register H one place to the left, with a 0 transferred into the least significant bit of H and the most significant bit of H transferring into the carry bit, you can use the instruction SLA s. The required op-code obtained from Table 10.7 is

```
CB
24
```

The comment in Table 10.7, that the required instruction format and states are as shown for the RLC r instructions, defines the first byte as being CB and the second byte as 00100100, ie 24.

Bit Addressing

It is possible to address a single bit within a selected register or RAM location and set or reset the bit according to the specified instruction.

The following examples illustrate this form of addressing. Note that the bits in a byte are numbered as 0, 1, 2, . . . , 6 and 7, where bit 0 is the least significant bit and bit 7 is the most significant bit.

1. The instruction BIT b, r (given in Table 10.8) will set the Z flag equal to the complement of the logic state of the addressed bit, b, in the selected r register. For example, to set the Z flag equal to the complement of bit 3 in the L register the required op-code is

CB
5D

2. The instruction RES b,(HL) is used to reset bit b in the RAM location addressed by the content of the HL register pair, see Table 10.8. For example, the instruction

CB
AE (10 101 110)

will reset bit 5 in the RAM location addressed by the content of the HL register pair at the time of executing the instruction.

Indexed Addressing

The Z80A microprocessor uses this form of memory addressing to access the desired data byte to be used with the instruction. It sets up the required memory address by adding the content of the selected index register, IX or IY, to a *displacement byte*, d, with the index register content and displacement byte unchanged when the instruction is executed.

The displacement byte is a two's complement eight-bit binary number (see Chapter 2), and therefore it is possible for this byte to have a value in the range +127(01111111) to −128(10000000) inclusive. Consequently memory locations are addressed using (IX + d) or (IY + d), where $-128 \leqslant d \leqslant 127$.

The instruction LD r, (IX + d) given in Table 10.1 will now be used to illustrate this mode of addressing. For this example suppose that the content of the index register IX is Ø2ØØ and that it is required that the data held in memory location Ø1FB is to be copied (loaded) into register E. The required instruction (see Table 10.1) is

DD
5E (01 011 110)
FB (11111011 = −5 = Ø1FB−Ø2ØØ)

Relative addressing

This form of addressing mode applies to some of the Z80A jump instructions, which are formed by the appropriate op-code followed by a two's complement displacement byte denoted by e–2.

The microprocessor achieves the desired change (jump) in program control by modifying the content of the program counter, PC, thereby enabling access to the next desired program instruction.

The two's complement binary number equal to 'e' is added to the memory address of the jump instruction op-code (held in PC) to achieve the desired modification to PC. Since e–2 is a two's complement eight-bit binary number representing numbers in the range $-128 \leqslant (e-2) \leqslant 127$, then it follows that program control may be made to jump to any location in the range –126 to 129 from the address of the jump instruction op-code. This means that because program control does not jump to a specified memory address it may be possible to easily relocate the program in another part of memory. It is better, therefore, to use relative addressing mode instructions rather than extended addressing mode instructions. The following example uses the instruction JR e, given in Table 10.9, to illustrate this mode of addressing.

Suppose that the required instruction op-code, 18, is stored in memory location 4AFC and we require the instruction to make program control jump to memory location 4BØ2, then the necessary machine code instruction, obtained from Table 10.9, is

18	Op-code
Ø4	e–2 (e = 6; 4AFC + 6 = 4BØ2)

In our consideration of the addressing modes we have shown the machine codes for selected instructions. When you store these types of instructions in sequential memory locations you have created a machine code program which the microprocessor runs by fetching and executing each instruction in turn.

As a simple illustration of a machine code program, listed below are the codes for a program that loads accumulator A with the hexadecimal number 8D, then copies the content of A into register B, then forms the one's complement of the content of A, and finally copies the content of A into register C.

3E	Op-code	LD A,8D	Table 10.1
8D	Data		
47	Op-code	LD B,A	Table 10.1
2F	Op-code	CPL	Table 10.5
4F	Op-code	LD C,A	Table 10.1

Implementation of these four instructions results in (A) = 72, (B) = 8D and (C) = 72, where ( ) indicates register content.

Storing and running machine code programs

Two places in memory where you can store a machine code program are in the same area as your Basic program or in a separate area created for this purpose between RAMTOP and user defined graphics (see Fig. 9.11).

To include machine code within a Basic program you can use an appropriate **REM** statement. However, from a practical point of view, the choice of where to place the **REM** statement is governed by the need to know the address of the first byte of the machine code program. When the **REM** is the first line of your Basic program, the address of the first byte used for the Basic program is determined by peeking into the *system variable* PROG (see Appendix C). This can be done with

```
10 LET x = PEEK 23635
15 LET y = PEEK 23636
20 PRINT "Address of First Byte of Basic " ' " is "; x + 256*y
```

The address of the first machine-code byte will be the address obtained using the above program plus five, because the line number and **REM** token require four and one memory locations respectively. You will find that the address of the first byte of your Basic program is 23755, hence the address of the first byte of machine code will be 23760. If you place the **REM** statement elsewhere in your program you will have to determine the address in RAM which can hold the first byte of the machine code program, and this is not an easy task.

When using the **REM** statement in the first line of your Basic program the number of characters between the **REM** token and **ENTER** corresponds to the number of bytes that can be used for machine code.

The method of entering machine code in a **REM** statement uses a two-pass operation. In the first pass the **REM** statement is used to reserve the required memory space for the machine code program, and in the second pass the machine code bytes are converted to their decimal equivalent code and poked into the appropriate memory locations.

Let us consider how we can enter the following three bytes of machine code

```
04      INC B
0D      DEC C
C9      RET
```

The first pass operation to reserve the required three memory bytes is achieved by entering

```
1 REM bbb
```

where the three characters bbb correspond to the memory bytes for the machine code. The bytes of the machine code must now be converted to their decimal equivalents (see Table 7.1), that is

Ø4	converts to Ø4
ØD	converts to 13
C9	converts to 2Ø1

and subsequently poked into memory addresses 23760, 23761 and 23762. Note that on completing the above two-pass operation your **REM** statement line will be listed as

1 **REM ?**

When your machine code is included in a **REM** statement it is part of a Basic program and is subject to all the Basic commands, eg **EDIT, LIST, SAVE, NEW** (which erases the entire program), etc.

A machine code program is linked to a Basic program by using the **USR** function. The memory address of the first byte of machine code is the number used after the **USR** function. The hexadecimal equivalent of this number is loaded into the BC register pair of the Z80A and, after executing the last machine code instruction, ie the essential return instruction (C9), the content of the BC register pair returns to the **USR** function.

Therefore to access and run the three bytes of machine program listed above once they have been entered, you can use

PRINT USR 2376Ø

followed by **ENTER**. The displayed result is 24015, which is the correct result. That is, before the machine code instructions are executed, the values stored in the B and C registers of the Z80A microprocessor are

(B) = 92 and (C) = 208

(recall that 23760 = (92 × 256) + 208), and then after executing the first instruction, INCB, the register contents are

(B) = 93 and (C) = 208,

then after executing the second instruction, DEC C, the register contents are

(B) = 93 and (C) = 207

and then after the return instruction (RET) the **USR** function returns with the value

(93 × 256) + 207 = 24015

We have seen in the example above that the **USR** function is used in a line of a Basic program to access the machine code program, and

you should note that if you use

25 **LET** x = **USR** 2376Ø

then, after executing the machine code program above, the value of x is set equal to 24015.

Note also that a return instruction (RET) must be included as the last op-code in a machine code program. Otherwise it is quite possible that your ZX Spectrum will fetch and execute the codes which exist in the RAM locations following the end of your machine code program and, since these are unspecified, then a 'crash' condition may result. That is, the ZX Spectrum operating system loses control and makes the microcomputer incapable of doing anything useful. For example, you will recognise this condition when you are unable to input control commands from the keyboard. Unfortunately the only way to terminate a crash condition is momentarily to disconnect the 9V DC power supply, which of course results in the loss of your program in RAM.

The alternative area in RAM where you can store a machine code program is an area which you create above RAMTOP but below the user defined graphics area. This area is always used when you do not wish to have your machine code program erased by **NEW.**

In the Spectrum the user defined graphics area occupies the top 168 bytes of RAM, and RAMTOP is normally the next address below this (see Fig. 9.11). Therefore in the 16K Spectrum RAMTOP is normally at address 32599, while in the 48K Spectrum it is at 65367. You can redefine the address of RAMTOP by putting the desired RAMTOP value in a **CLEAR** statement using the form

CLEAR Desired RAMTOP value

Note that **CLEAR** also clears all the program variables, display file (CLS) and **GOSUB** stack, the latter being put at the new RAMTOP, and it also does a **RESTORE** and resets the **PLOT** position.

When you have redefined RAMTOP to reserve sufficient memory bytes for your machine code program above RAMTOP and below the user defined graphics area, you may poke your machine code into the available memory area. The machine code program is then accessed by using the **USR** function followed by the start address of the machine code program, as described earlier in this section.

For example, consider using a 48K Spectrum to run the three bytes of machine code which we considered above, ie

Ø4	INC B
ØD	DEC C
C9	RET

We shall assume that the first byte of machine code is to be stored at memory address 6ØØØ1. So we may redefine RAMTOP by entering

CLEAR 6ØØØØ

Next we convert the machine codes to their decimal equivalents, which in this case, as we saw earlier in this section, gives

Ø4	INC B
13	DEC C
2Ø1	RET

These values are then poked into the memory addresses 6ØØØ1, 6ØØØ2, 6ØØØ3 respectively, and the program can be run by entering

PRINT USR 6ØØØ1

You may note, assuming that you have access to a 48K machine, that the Spectrum displays 6Ø256, which is the correct result. That is, before the machine code instructions are executed the values stored in the B and C registers are

(B) = 234 and (C) = 97
(recall 6ØØØ1 = (234 × 256) + 97)

and after executing the INC B, DEC C and RET instructions the **USR** function returns with the value

(235 × 256) + 96 = 6Ø256

Exercise
In your 16K or 48K Spectrum define RAMTOP to be 3ØØØØ and in the next three successive addresses enter the machine codes

Ø4	INC B
ØD	DEC C
C9	RET

and run the program. Verify that after running the program the **USR** function returns with the value 3Ø256

Note that once you have redefined RAMTOP it stays at the redefined value until you change RAMTOP to a new value or remove the d.c. power supply.

You can store your machine code programs on cassette tape by using the **SAVE** statement in the form

SAVE "name of machine code program" **CODE** start address, number of bytes in program

For example, for the exercise above we can use

SAVE "simple program" **CODE** 3ØØØ1,3

In this case we have named the three bytes "simple program".

The tedium of converting machine code program bytes to their decimal equivalent and then poking them into memory locations

above RAMTOP can be eliminated by using Program 6 in Appendix A.

Illustrative examples

Pseudo PRINT AT program

The main purpose of this machine code programming example, which implements a pseudo PRINT AT x,y operation, is:

(a) to demonstrate how the main register set of the Z80A microprocessor may be used with various forms of addressing mode; and

(b) to show how a character stored in ROM may be copied in the Display File, and how the character attribute may be defined and placed in the Attribute File (all dealt with in Chapter 7).

The flowchart for this example is shown in Fig. 10.7. The first part of the process involves setting the content of the HL register pair equal to the Attribute File Address, that is

(HL) = 22528 + y + (32*x)

where x and y correspond to the character- grid row and column numbers respectively (see Fig. 7.6). Accumulator A is then loaded with the character attribute, and this is stored in the Character Attribute File byte, which is addressed using the content of the HL register pair. At this point in the process the character attribute has been defined.

The next part of the process involves setting the content of the BC register pair equal to the memory address of the first Display File byte of the character to be displayed, and this is determined using the equation given in Chapter 7, that is

(BC) = 16384 + y + x (32*r) + (256*character byte number) + (2048*screen section number)

where r is the character row number in the range $0 \leqslant r \leqslant 7$ (see Fig. 7.10). The character byte number is equal to zero for the first Display File byte, and the screen section number is determined by the value of x; therefore it follows that

(BC) = 16384 + y + (32*x) for (x = r) <8, or
(BC) = 18432 + y + (32*(x−8)) for r = (x−8) and $8 \leqslant x \leqslant 15$, or
(BC) = 20480 + y + (32*(x−16)) for r = (x−16) and x > 15

The flowchart shows that the address of the first Display File byte, addressed by (BC), is then saved in the IY Index Register. This releases

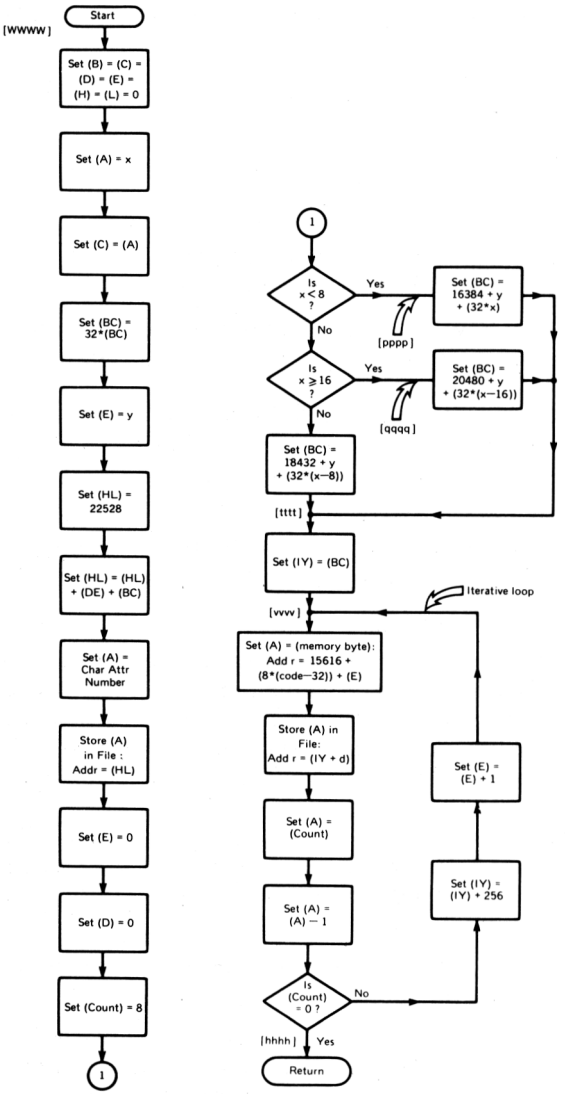


Fig. 10.7. Flowchart for a pseudo PRINT AT program

the BC register pair, which is then used to hold the memory address of the first byte of the character stored in ROM, which is specified by its Character Set code. That is

(BC) = 15616 + (8*(code−32)) + (E)

where (E) = k gives access to the k_{th} byte of the character for $0 \leqslant k \leqslant 7$. The first accessed byte of the ROM character is loaded into Accumulator A, and then the content of A,(A), is stored in the Display File byte which is addressed using the previously established content of Index Register, IY. At this point in the process the first byte of the ROM character has been copied into the first byte of the Display File character.

On entering the iterative loop for the first time the count variable is non-zero, and therefore the iterative part of the process is repeated. However, in order to gain access to the next ROM character byte, and the next byte of the Display File character, it is necessary to update (E) and (IY) respectively, as shown in the flowchart. The count variable is decremented each time the iterative loop is implemented, and consequently the program counts the number of ROM character bytes copied into the Display File. When the eight bytes of the ROM character have been copied the essential Return instruction is executed.

The machine code program for this process is listed below. Before using the program you should note the following points:

(a) it is convenient to express the program memory addresses relative to RAMTOP, RT, and

(b) it is necessary to initialise the program variables x,y, code and Char. Attr. Number, by poking the required numbers into their respective memory locations.

(c) Valid ranges are: $0 \leqslant x \leqslant 21$, $0 \leqslant y \leqslant 31$ and $32 \leqslant$ code $\leqslant 127$.

Considering point (a) above, note that if the program variables and the program, as listed below, are loaded immediately above RAMTOP, then the general addresses used in the listing would have values relative to RAMTOP, as follows:

mmmm	RAMTOP + 1, ie RT + 1
aaaa	RT + 2
bbbb	RT + 3
cccc	RT + 4
dddd	RT + 5
wwww	RT + 6
tttt	RT + 109
vvvv	RT + 115
pppp	RT + 166
qqqq	RT + 196
hhhh	RT + 228

For the 16K or 48K Spectrum, when RT is made equal to 32300 the memory addresses for the program are

	Decimal	Hexadecimal
mmmm	323Ø1	7E2D
aaaa	323Ø2	7E2E
bbbb	323Ø3	7E2F
cccc	323Ø4	7E3Ø
dddd	323Ø5	7E31
wwww	323Ø6	7E32
tttt	324Ø9	7E99
vvvv	32415	7E9F
pppp	32466	7ED2
qqqq	32496	7EFØ
hhhh	32528	7F1Ø

Program listing

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
mmmm	Count	—	Program variable	—
aaaa	x	—	Program variable	—
bbbb	y	—	Program variable	—
cccc	Code	—	Program variable	—
dddd	Char. Attr. Number	—	Program variable	—
wwww	Ø6	LDr,n	(B) = Ø	10.1
	ØØ			
	ØE	LDr,n	(C) = Ø	10.1
	ØØ			
	16	LDr,n	(D) = Ø	10.1
	ØØ			
	1E	LDr,n	(E) = Ø	10.1
	ØØ			
	26	LDr,n	(H) = Ø	10.1
	ØØ			
	2E	LDr,n	(L) = Ø	10.1
	ØØ			
	3A	LD A,(nn)		10.1
	aa (LO byte of Addr;n)		(A) = x	
	aa (HI byte of Addr;n)			
	4F	LDr,s	(C) = x, ie (BC) = x	10.1
	CB	SLA s		10.7
	21		2*(BC)	
	CB	RLs		10.7
	1Ø			
	CB	SLAs		10.7
	21		4*(BC)	
	CB	RLs		10.7
	1Ø			
	CB	SLAs		10.7
	21		8*(BC)	
	CB	RLs		10.7
	1Ø			
	CB	SLAs		10.7
	21		16*(BC)	
	CB	RLs		10.7
	1Ø			
	CB	SLAs		10.7
	21		32*(BC)	
	CB	RLs		10.7
	1Ø			

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
	3A	LD A,(nn)		10.1
	bb (LO byte of Addr:n)		(A) = y	
	bb (HI byte of Addr:n)			
	5F	LD r,s	(E) = y	10.1
	21	LD dd,nn		10.2
	00 (LO data byte)		(HL) = 5800 (hex)	
	58 (HI data byte)		= 22528 (decimal)	
	19	ADD HL,ss	(HL) = (HL) + (DE)	10.6
	09	ADD HL,ss	(HL) = (HL) + (BC)	10.6
	3A	LD A,(nn)		10.1
	dd (LO byte of Addr:n)			
	dd (HI byte of Addr:n)			
	77	LD(HL),r	Store (A):Addr = (HL)	10.1
	1E	LD r,n	(E) = 0	10.1
	00			
	16	LD r,n	(D) = 0	10.1
	00			
	3E	LD r,n	(A) = 8	10.1
	08			
	32	LD(nn),A		10.1
	mm (LO byte of Addr:n)		(Count) = 8	
	mm (HI byte of Addr:n)			
	3A	LD A,(nn)		10.1
	aa (LO byte of Addr:n)		(A) = x	
	aa (HI byte of Addr:n)			
	C6	ADD A,n	(A) = (A) − 8	10.4
	F8	(= −8)		
	FA	JP cc,nn	If x < 8 Jump	10.9
	pp (LO byte of Addr:n)		to pppp, otherwise	
	pp (HI byte of Addr:n)		continue	
	3A	LD A,(nn)		10.1
	aa (LO byte of Addr:n)		(A) = x	
	aa (HI byte of Addr:n)			
	C6	ADD A,n	(A) = (A) — 16	10.4
	F0	(= −16)		
	F2	JP cc,nn	If x >15 Jump	10.9
	qq (LO byte of Addr:n		to qqqq, otherwise	
	qq (HI byte of Addr:n)		continue	
	3A	LD A,(nn)		10.1
	aa (LO byte of Addr:n)		(A) = x	
	aa (HI byte of Addr:n)			
	C6	ADD A,n	(A) = x−8	10.4
	F8	(= −8)		
	CB	SLA r	(A) = 2*(x−8)	10.7
	27			
	CB	SLA r	(A) = 4*(x−8)	10.7
	27			
	CB	SLA r	(A) = 8*(x−8)	10.7
	27			
	CB	SLA r	(A) = 16*(x−8)	10.7
	27			
	CB	SLA r	(A) = 32*(x−8)	10.7
	27			
	21	LD dd,nn		10.2
	00 (LO data byte)		(HL) = 4800 (hex)	
	48 (HI data byte)		= 18432 (decimal)	
	06	LD r,n	(B) = 0	10.1
	00			
	4F	LD r,s	(C) = (A)	10.1
	09	ADD HL,ss	(HL) = (HL) + (BC)	10.6
	3A	LDA,(nn)		10.1
	bb (LO byte of Addr:n)		(A) = y	
	bb (HI byte of Addr:n)			

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
	4F	LD r,s	(C) = y	10.1
	Ø9	Add HL,ss	(HL) = (HL) + (BC)	10.6
	44	LD r,s	(B) = (H)	10.1
	4D	LD r,s	(C) = (L) } (BC) = (HL)	10.1
tttt	FD	LD IY,nn		10.2
	21		(IY) = Ø	
	ØØ			
	ØØ			
	FD	ADD IY,rr	(IY) = (IY) + (BC)	10.6
	Ø9		= Addr of Disply File byte	
vvvv	26	LD r,n	(H) = Ø	10.1
	ØØ			
	3A	LDA,(nn)		10.1
	cc (LO byte of Addr:n)		(A) = code	
	cc (HI byte of Addr:n)			
	C6	Add A,n	(A) = code −32	10.4
	EØ	(= −32)		
	6F	LD r,s	(L) = code −32	10.1
	CB	SLA,r	(L) = 2*(code −32)	10.7
	25			
	CB	RL,r	(HL) = 2*(code −32)	10.7
	14			
	CB	SLA,r	(L) = 4*(code −32)	10.7
	25			
	CB	RL,r	(HL) = 4*(code −32)	10.7
	14			
	CB	SLA,r	(L) = 8*(code −32)	10.7
	25			
	CB	RL,r	(HL) = 8*(code −32)	10.7
	14			
	19	ADD HL,ss	(HL) = (HL) + (DE)	10.6
	Ø1	LD dd,nn		10.2
	ØØ (LO data byte)		(BC) = 3D00 (hex)	
	3D (HI data byte)		= 15616 (decimal)	
	Ø9	ADD HL,ss	(HL) = 15616 + 8*(code−32)	
			+ (E)	10.6
	7E	LD r,(HL)	(A) = (memory byte)	
			:Addr = (HL)	10.1
	FD	LD(IY + d),r		10.1
	77		Store (A):Addr = (IY + d)	
	ØØ	d		
	3A	LDA,(nn)		10.1
	mm (LO byte of Addr:n)		(A) = (Count)	
	mm (HI byte of Addr:n)			
	3D	DECs	(A) = (A)—1	10.4
	CA	JP cc,nn	If (A) = 0 Jump	10.9
	hh (LO byte of Addr:n)		to hhhh, otherwise	
	hh (HI byte of Addr:n)		continue	
	32	LD(nn),A		10.1
	mm (LO byte of Addr:n)		(Count) = (A)	
	mm (HI byte of Addr:n)		ie (Count) = (Count)—1	
	Ø6	LD r,n	(B) = 0 } (BC) = 255	10.1
	ØØ			
	ØE	LDr,n	(C) = FF	10.1
	FF			
	FD	INC IY	(IY) = (IY) + 1	10.6
	23			
	FD	ADD IY,rr	(IY) = (IY) + (BC)	10.6
	Ø9		ie (IY) = (IY) + 256	
	1C	INC r	(E) = (E) + 1	10.4

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
	C3	JP nn		10.9
	vv (LO byte of Addr:n)		Unconditional	
	vv (HI byte of Addr:n)		Jump to vvvv	
pppp	3A	LDA,(nn)		10.1
	aa (LO byte of Addr:n)		(A) = x	
	aa (HI byte of Addr:n)			
	CB	SLA r	(A) = 2*x	10.7
	27			
	CB	SLA r	(A) = 4*x	10.7
	27			
	CB	SLA r	(A) = 8*x	10.7
	27			
	CB	SLA r	(A) = 16*x	10.7
	27			
	CB	SLA r	(A) = 32*x	10.7
	27			
	21	LD dd,nn		10.2
	ØØ (LO data byte)		(HL) = 4000 (hex)	
	4Ø (HI data byte)		= 16384 (decimal)	
	Ø6	LD r,n	(B) = Ø	10.1
	ØØ			
	4F	LD r,s	(C) = (A)	10.1
	Ø9	ADD HL,ss	(HL) = (HL) + (BC)	10.6
	3A	LDA,(nn)		10.1
	bb (LO byte of Addr:n)		(A) = y	
	bb (HI byte of Addr:n)			
	4F	LD r,s	(C) = y	10.1
	Ø9	ADD HL,ss	(HL) = (HL) + (BC)	10.6
	44	LD r,s	(B) = (H) } (BC) = (HL)	10.1
	4D	LD r,s	(C) = (L) } = 16384	
			+ y + 32*x	10.1
	C3	JP nn		10.9
	tt (LO byte of Addr:n)		Unconditional	
	tt (HI byte of Addr:n)		Jump to tttt	
qqqq	3A	LDA,(nn)		10.1
	aa (LO byte of Addr:n)		(A) = x	
	aa (HI byte of Addr:n)			
	C6	ADD A,n	(A) = x—16	10.4
	FØ	(= —16)		
	CB	SLA r	(A) = 2*(x—16)	10.7
	27			
	CB	SLA r	(A) = 4*(x—16)	10.7
	27			
	CB	SLA r	(A) = 8*(x—16)	10.7
	27			
	CB	SLA r	(A) = 16*(x—16)	10.7
	27			
	CB	SLA r	(A) = 32*(x—16)	10.7
	27			
	21	LD dd,nn		10.2
	ØØ (LO data byte)		(HL) = 5000 (hex)	
	5Ø (HI data byte)		= 20480 (decimal)	
	Ø6	LD r,n	(B) = Ø	10.1
	ØØ			
	4F	LD r,s	(C) = 32*(x—16)	10.7
	Ø9	ADD HL,ss	(HL) = (HL) + (BC)	10.6
	3A	LDA,(nn)		10.1
	bb (LO byte of Addr:n)		(A) = y	
	bb (HI byte of Addr:n)			
	4F	LD r,s	(C) = y	10.1

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
	09	ADD HL,ss	(HL) = (HL) + (BC)	10.6
	44	LD r,s	(B) = (H) } (BC) = (HL)	10.1
	4D	LD r,s	(C) = (L) } = 20480 + y	
			+ 32*(x—16)	10.1
	C3	JP nn		10.9
	tt (LO byte of Addr:n)		Unconditional	
	tt (HI byte of Addr:n)		Jump to tttt	
hhhh	C9	RET	Return	10.10

Exercise

Using Program 6, Appendix A, input a suitable RAMTOP value and load the above machine code program. Remember to use the correct hexadecimal addresses relative to RAMTOP and enter them at the appropriate points in the program listing. Initialise the values of x, y, code and Character Attribute Number, and then run the machine code program using

5 **GO TO USR** RAMTOP + 6
7 **STOP**

where RAMTOP is the numerical value you used in the loading program.

IN/OUT program

The main purpose of this machine code programming example is to demonstrate how the Z80A microprocessor can be used to implement a data byte IN/OUT operation at the machine code level. This is achieved by using the one-byte memory-mapped interface described in Chapter 9. Furthermore, in this example, the microprocessor is used to determine the number of bits in the input data byte set equal to 1, and it then outputs the result as a binary number through the I/O port.

The flowchart for this example is shown in Fig. 10.8. The first part of the process involves setting the content of the BC register pair equal to the address of the one-byte memory-mapped interface. The next step in the process is used to input the interface data byte into Accumulator A. Register D is used as a counter, and its initial value is set equal to zero. The arithmetic shift left instruction moves the content of Accumulator A one place left, with the most significant bit of the data byte moving into the carry bit position and the least significant bit being set equal to zero. Consequently, if this action sets the C flag equal to 1 the counter is incremented by one, otherwise the incrementing operation is skipped. When the content of Accumulator A is zero, ie none of its bits is set equal to 1, the shifting process is terminated and the binary count value held in register D is displayed using the interface LEDs.

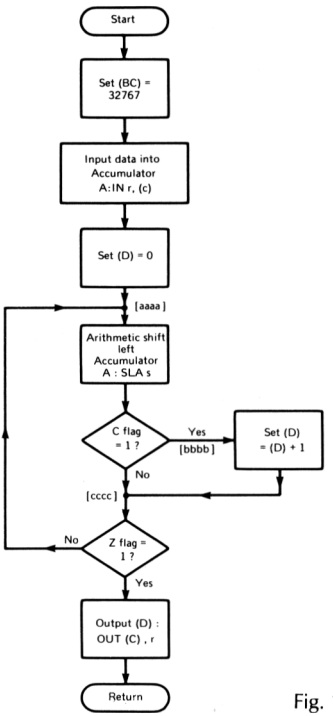


Fig. 10.8. Flowchart for a machine code I/O program

The machine code program for this process is listed below; note that it is convenient to express the program memory addresses relative to RAMTOP, RT. The general addresses used in the listing have the following relative values:

aaaa	RT + 9
bbbb	RT + 20
cccc	RT + 14

Note that for IN/OUT operations the interface selector switch, SW, must be set to position 2 (see Fig. 9.15).

Program listing

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
RT + 1	06	LD r,n	(B) = 7F	10.1
	7F		(BC) = 32767	
	0E	LD r,n	(C) = FF	10.1
	FF			
	ED	IN r,(C)	(A) = Input data	10.11
	78			
	16	LD r,n	(D) = 0	10.1
	00			

Address	*Memory byte*	*Mnemonic*	*Remarks*	*Instruction Set Table*
aaaa	CB	SLAs	2*(A)	10.7
	27			
	DA	JP cc,nn	If C = 1 Jump to	10.9
	bb	LO byte of Addr	bbbb, otherwise	
	bb	HI byte of Addr	continue	
cccc	C2	JP cc,nn	If Z = Ø Jump to	10.9
	aa	LO byte of Addr	aaaa, otherwise	
	aa	HI byte of Addr	continue	
	ED	OUT (C),r	Output result	10.11
	51			
	C9	RET	Return	10.9
bbbb	14	INC r	(D) = (D) + 1	10.4
	C3	JP nn		10.9
	cc	LO byte of Addr	Unconditional	
	cc	HI byte of Addr	Jump to cccc	

Specifically, when RAMTOP, RT, is made equal to 32300, the memory addresses for the program above are

	Decimal	Hexadecimal
aaaa	32309	7E35
bbbb	32314	7E3A
cccc	32320	7E40

Exercise

Using Program 6, Appendix A, input a suitable RAMTOP value and load the above machine code program. Remember to use the correct hexadecimal addresses relative to RAMTOP, and enter them at the appropriate points in the program listing. Run the program using

```
2ØØ LET A = USR RAMTOP + 1
21Ø GO TO 2ØØ
```

where RAMTOP is the numerical value you used in the loading program.

Sound generating routine: CALL Ø3B5

A machine-code program can access the Basic routines in the ROM by including the CALL instruction with the address of the routine to be executed. The sound generating routine, for instance, is accessed using CALL Ø3B5.

Before calling this routine it is necessary to load the DE register pair with the number which determines the duration of the generated note, and the HL register pair must be loaded with the number which determines the pitch of the note.

The following ten bytes of machine code will implement this routine.

```
RAMTOP + 1   11      LD DE,nn
               n   LO byte of duration number
               n   HI byte of duration number
             21      LD, HL,nn
RAMTOP + 5     n   LO byte of pitch number
               n   HI byte of pitch number
             CD      CALL Ø3B5
             B5   LO byte of Routine Addr
             Ø3   HI byte of Routine Addr
RAMTOP + 10  C9      RET
```

Exercise
Using Program 6, Appendix A, input a suitable RAMTOP value and load the above machine code program. Use Ø5FF for the duration number and ØØC8 for the pitch number. Run the machine code program using

```
135 LET x = USR RAMTOP + 1
```

where RAMTOP is the numerical value used in the loading program.

Enter and run the following program and observe the effect of changing the pitch of the generated note.

```
3ØØ POKE RAMTOP + 5, 1ØØ
35Ø LET x = USR RAMTOP + 1
375 POKE RAMTOP + 5, 255
4ØØ LET x = USR RAMTOP + 1
425 GO TO 3ØØ
```

Concluding remarks

In this chapter the basic concepts of programming the Z80A microprocessor have been presented using machine code programming techniques. We have examined the addressing modes and available Z80A instructions and demonstrated their use by including some simple examples. We recommend that you study the illustrative examples, which have been included to assist in understanding the main register set of the Z80A microprocessor, its addressing modes, and how to implement I/O data transfer.

Appendix A

Programs

Program 1: Number conversion

This program may be used to convert positive integer binary and hexadecimal numbers to their equivalent decimal form. It may also be used to convert positive integer decimal numbers to a twenty-five digit binary or hexadecimal number (to convert to binary the maximum decimal number is 33 554 431 and for hexadecimal the maximum decimal number is approximately 1.2676506E + 30).

```
 2Ø CLS
 25 PRINT "This program converts:
           1. Binary to Decimal
           2. Hexadecimal to Decimal
           3. Decimal to Binary
           4. Decimal to Hexadecimal
           Input 1, 2, 3 or 4 to select"
 45 INPUT N
 55 IF N = 1 THEN GO TO 1Ø5
 65 IF N = 2 THEN GO TO 2Ø5
 75 IF N = 3 THEN GO TO 3ØØ
 85 IF N = 4 THEN GO TO 4Ø5
 95 GO TO 2Ø
1Ø5 PRINT
1Ø8 PRINT "Input Binary number"
1Ø9 LET radix = 2
11Ø INPUT B$
115 LET P = 1
12Ø LET R = Ø
125 LET L = LEN B$
13Ø FOR K = L TO 1 STEP -1
131 LET j = Ø
133 IF CODE B$(K) > 57 THEN LET j = 7
```

```
135 LET a = (CODE (B$(K)) –48–j)*P
145 LET P = P*radix
155 LET R = R + a
165 NEXT K
175 PRINT
176 IF radix = 16 THEN GO TO 225
178 PRINT "The Binary number" 'B$' " converts to ";R
200 GO TO 500
205 PRINT
208 PRINT "Input Hexadecimal number"
209 LET radix = 16
210 GO TO 110
225 PRINT "The Hexadecimal number" 'B$' " converts to ";R
230 GO TO 500
300 LET b = 2
305 DIM f$(25)
306 PRINT
307 PRINT "Input Decimal number"
310 INPUT S: LET d = S
315 FOR i = 1 TO 25
320 LET x = d/b
325 LET r = d – (INT (x)*b)
326 IF r > 9 THEN GO TO 420
330 LET f$(i) = CHR$ (48 + r)
335 LET d = INT (x)
345 NEXT i
355 PRINT
365 PRINT "The Decimal number" 'S' " converts to "
375 FOR i = 25 TO 1 STEP –1
385 PRINT f$(i);
395 NEXT i
400 GO TO 500
405 LET b = 16
410 GO TO 305
420 LET r = r + 7: GO TO 330
500 PRINT : PRINT
520 PRINT "Enter G to GO AGAIN or S to STOP"
555 INPUT K$
565 IF K$ = "S" THEN STOP
666 IF K$ = "G" THEN GO TO 20
```

Program 2: Stop-watch and elapsed time indicator

This program implements a 24-hour stop-watch and elapsed time indicator. You input the initial starting time and then press S to start

the watch and hold down the H key to stop it. The time in hours, minutes and seconds is displayed throughout the timing period. The difference between the initial time and the final time (the elapsed time) is displayed on stopping the watch.

```
  3 DIM a$(7)
  5 PRINT "Input Hours Setting"
 15 INPUT h: PRINT AT Ø,22; "=" ; h
 18 LET hØ = h
 25 PRINT "Input Minutes Setting"
 35 INPUT m: PRINT AT 1, 22; "="; m
 37 LET mØ = m
 45 PRINT "Input Seconds Setting"
 55 INPUT sec: PRINT AT 2,22; "="; sec
 57 LET secØ = sec
 6Ø IF NOT (h < =23 AND m < = 59 AND sec < = 59) THEN GO
    TO 4ØØ
 64 PRINT '"Press Key S to Start Watch" ' ' "To Halt Watch Hold
    Down Key H"
 66 IF INKEY$ < > "S" THEN GO TO 66
 75 CLS
 95 PRINT "Hours     Minutes     Seconds"
1Ø5 LET X = 16
115 LET S = X
145 LET Y = 3
155 LET X = X –1
165 IF X < > Ø THEN GO TO 195
175 LET X = S
185 LET Y = Y – 1
195 IF Y < > Ø THEN GO TO 155
2ØØ PAUSE 2
2Ø2 LET sec = sec + 1
2Ø5 IF sec = 6Ø THEN LET m = m + 1
21Ø IF sec = 6Ø THEN PRINT AT 1,18;a$
215 IF sec = 6Ø THEN LET sec = Ø
225 IF m = 6Ø THEN LET h = h + 1
23Ø IF m = 6Ø THEN PRINT AT 1,8;a$
235 IF m = 6Ø THEN LET m = Ø
24Ø IF h = 24 THEN PRINT AT 1,Ø;a$
245 IF h = 24 THEN LET h = Ø
335 LET a = 1
34Ø IF h < 1Ø THEN LET a = 2
342 PRINT AT 1,a;h
343 LET b = 1Ø
346 IF m < 1Ø THEN LET b = 11
347 PRINT AT 1,b;m
```

```
349 LET c = 2Ø
351 IF sec < 1Ø THEN LET c = 21
353 PRINT AT 1,c;sec
36Ø IF INKEY$ = "H" THEN GO TO 365
363 GO TO 1Ø5
365 PRINT AT 1Ø,Ø; "Release Key H"
37Ø IF INKEY$ < > "" "THEN GO TO 37Ø
375 GO SUB 5ØØ
38Ø STOP
4ØØ CLS
41Ø GO TO 5
5ØØ LET hØ = h - hØ
51Ø LET mØ = m - mØ
515 LET secØ = sec - secØ
52Ø LET t = hØ*36ØØ + mØ*6Ø + secØ
54Ø LET h1 = INT (t/36ØØ)
545 LET m1 = INT ((t - h1*36ØØ)/6Ø)
555 LET sec1 = t - m1*6Ø - h1*36ØØ
565 PRINT AT 5,Ø; "Elapsed Time is"
57Ø LET a = 1
572 IF h1 < 1Ø THEN LET a = 2
575 PRINT AT 7,a;h1
577 LET b = 1Ø
585 IF m1 < 1Ø THEN LET b = 11
59Ø PRINT AT 7,b;m1
6ØØ LET c = 2Ø
61Ø IF sec1 < 1Ø THEN LET c = 21
62Ø PRINT AT 7,c;sec1
63Ø RETURN
```

Note that in the above program the basic time period used in the stop watch is only approximately one second. Adjustment to this period can be made by varying the duration of the pause in line 200 of the program, or by varying the values of X and Y in lines 105 and 145 respectively. For example, try X = 15, Y = 3 and **PAUSE** 1.

Program 3: Displaying characters in ROM

In Chapter 7 the form of stored characters in ROM was described, and in Plate 7.3 the 8 × 8 binary bit pattern and corresponding 8 × 8 pixel grid definition, magnified by a factor of 64, for the © character was given. The program listed below was used to produce this result.

Character codes in the range 32 to 127 inclusive (see Table 7.1) may be used with this program, to display the corresponding output in the form shown in Plate 7.3.

```
  4 PAPER 6
  5 INPUT code
 12 PRINT AT 5,6; "The Character is ";CHR$ code
 25 FOR x = 1 TO 8
 30 LET j = 15615 + 8*(code - 32) + x
 32 LET g = PEEK j
 40 GOSUB 150
 50 NEXT x
150 FOR n = 1 TO 8
155 LET y = g/2
165 LET r = g - (INT y*2)
175 LET a$ = CHR$ (48 + r)
177 PRINT AT 7 + x,12 - n;a$
178 IF a$ = "0" THEN LET a$ = "  "
179 IF a$ = "1" THEN LET a$ = " ■"
180 PRINT AT 7 + x, 28 - n;a$
185 LET g = INT y
195 NEXT n
200 IF x = 8 THEN PAPER 4
210 IF x = 8 THEN STOP
225 RETURN
```

The program can be modified to obtain a display of the 8 × 8 bit pattern and pixel grid definition of a character entered via the keyboard. In this case the keyboard entered character is converted to the corresponding code using the **INKEY$** function. To do this you must change lines 5, 12 and 30 in the program to

```
  5 IF INKEY$ " "THEN GO TO 5
 12 PRINT AT 5,6; "The Character is "; Z$
 30 LET j = 15615 + 8*(CODE Z$ - 32) + x
```

and insert the additional lines

```
  3 PAUSE 10
 10 LET Z$ = INKEY$
```

Note that the modified form of the program does not enable you to input all of the characters with codes in the range 32 to 127 inclusive. For example the } character (code 125) is excluded. However, the main advantage of using the modified form of the program is that it allows you to enter directly most characters without the need to look up the character codes in the Character Set summary (Table 7.1).

Program 4: ROM dump

This program can be used to print/display the hexadecimal codes stored in ROM. The value of 'a' defined in line 5 gives the first address

of ROM to be listed, and the value of 'a' defined in line 87 gives the last address of ROM to be listed. For the program listing below, with the values a = Ø (line 5) and a = 16384 (line 87), the complete ROM contents will be listed. To examine small sections of ROM simply redefine the start and finish addresses in lines 5 and 87 respectively.

In line 77 the listed display function is **LPRINT** so, if you do not have a ZX Printer, change the function to **PRINT** and the ROM contents will be displayed on the screen (see Plate A.4).

```
 5 LET a = Ø
 8 DIM h$(1,24)
15 LET d = Ø
17 LET x = PEEK a
2Ø FOR i = 2 TO 1 STEP -1
3Ø LET h = x/16
4Ø LET r = x - INT (h)*16
5Ø LET h$(1,i + d) = CHR$ (48 + r)
55 IF r > 9 THEN LET h$(1,i + d) = CHR$ (55 + r)
6Ø LET x = INT h
7Ø NEXT i
72 LET d = d + 3
77 IF d = 24 THEN LPRINT h$(1);" "; a - 7
8Ø IF d = 24 THEN GO TO 15
85 LET a = a + 1
87 IF a = 16384 THEN STOP
9Ø GO TO 17
```

On running the program you will see that the ROM contents are displayed in hexadecimal, eight successive locations being listed on one line, followed by the address of the first memory location for that line. A typical output listing for ROM addresses Ø to 1Ø5 is given below.

```
F3 AF 11 FF FF C3 CB 11 Ø
11 2A 5D 5C 22 5F 5C 18 7
18 43 C3 F2 15 FF FF FF 14
FF FF FF 2A 5D 5C 7E CD 21
CD 7D ØØ DØ CD 74 ØØ 18 28
18 F7 FF FF FF C3 5B 33 35
33 FF FF FF FF FF C5 2A 42
2A 61 5C E5 C3 9E 16 F5 49
F5 E5 2A 78 5C 23 22 78 56
78 5C 7C B5 2Ø Ø3 FD 34 63
34 4Ø C5 D5 CD BF Ø2 D1 7Ø
D1 C1 E1 F1 FB C9 E1 6E 77
6E FD 75 ØØ ED 7B 3D 5C 84
5C C3 C5 16 FF FF FF FF 91
FF FF FF FF F5 E5 2A BØ 98
```

Program 5: Find the treasure

This program is a game in which treasure has been buried in one character square of the screen character grid. You have to find the treasure by moving the ◇ character (graphics character d) upwards or downwards by pressing the u or d key respectively, and left or right by pressing the l or r key, until the ◇ character is on the character square where the treasure has been buried. The number of moves necessary to find the treasure is displayed and if you find the treasure in under ten moves you are an excellent treasure hunter.

Note that there are two blocks at the beginning of the program with line numbers 1 to 8. Which block you enter depends on whether you have the 16K or the 48K Spectrum.

```
1 POKE 32624,BIN ØØØØ1ØØØ
2 POKE 32625,BIN ØØØ1Ø1ØØ
3 POKE 32626,BIN ØØ1ØØØ1Ø
4 POKE 32627,BIN Ø1ØØØØØ1
5 POKE 32628,BIN ØØ1ØØØ1Ø
6 POKE 32629,BIN ØØØ1Ø1ØØ
7 POKE 32630,BIN ØØØØ1ØØØ
8 POKE 32631,BIN ØØØØØØØØ
```

16K Spectrum

```
1 POKE 65392, BIN ØØØØ1ØØØ
2 POKE 65393, BIN ØØØ1Ø1ØØ
3 POKE 65394, BIN ØØ1ØØØ1Ø
4 POKE 65395, BIN Ø1ØØØØØ1
5 POKE 65396, BIN ØØ1ØØØ1Ø
6 POKE 65397, BIN ØØØ1Ø1ØØ
7 POKE 65398, BIN ØØØØ1ØØØ
8 POKE 65399, BIN ØØØØØØØØ
```

48K Spectrum

```
1Ø BORDER 5: PAPER 6: INK Ø: FLASH Ø
12 LET n = Ø
14 CLS
25 LET r = INT (RND*2Ø)
35 LET c = INT (RND*31)
45 LET x = INT (RND*2Ø)
55 LET y = INT (RND*31)
6Ø IF x = r AND y = c THEN GO TO 45
65 PRINT AT x,y;"◇"
85 LET k$ = INKEY$
9Ø IF k$ = "" THEN GO TO 85
91 IF NOT (k$ = "u" OR k$ = "d" OR k$ = "l" OR k$ = "r")
   THEN GO TO 85
92 IF INKEY$ <>"" THEN GO TO 92
93 PRINT AT x,y;" "
```

```
 95 IF k$ = "u" THEN LET x = x - 1
115 IF x < 0 THEN LET x = 0
125 IF k$ = "d" THEN LET x = x + 1
135 IF x > 20 THEN LET x = 20
145 IF k$ = "l" THEN LET y = y - 1
155 IF y < 0 THEN LET y = 0
165 IF k$ = "r" THEN LET y = y + 1
175 IF y > 31 THEN LET y = 31
225 PRINT AT x,y;"◇"
235 IF x = r AND y = c THEN GO TO 350
240 LET n = n + 1
250 PRINT AT 21,0; "Number of Moves = ";n
260 GO TO 85
350 FOR g = 10 TO 0 STEP -1
355 BEEP .1,g
365 NEXT g
370 FLASH 1: PAPER 4
375 CLS : PRINT AT 10,10;"WELL DONE"
385 PRINT AT 12,3; "Treasure Found In ";n;" Moves"
390 PRINT AT 16,4; "Press Key G To Go Again"
400 PRINT AT 18,4; "Or Press Key S To Stop"
420 IF NOT (INKEY$ = "G" OR INKEY$ = "S") THEN GO TO
    420
425 IF INKEY$ = "G" THEN GO TO 10
430 IF INKEY$ = "S" THEN STOP
```

Program 6: Program to load machine code above RAMTOP

This program requires you to input, as a decimal number, the value of RAMTOP. This is set by using the **CLEAR** statement in line 75, which has the form

CLEAR Ramtop value

You then input each byte of your machine code program as two hexadecimal characters and these are stored in successive memory locations starting at the next address above the defined RAMTOP. When the last byte of your machine code program has been entered you enter Z to terminate the loading operation. To access and run the entered machine code program use the **USR** function followed by the start address which is, of course, RAMTOP plus one.

```
10 PRINT "Input Ramtop value"
20 INPUT RT
25 PRINT RT
```

```
28 LET A = RT
3Ø PRINT "Input Byte of machine code" ' "or Z to STOP"
35 INPUT H$
36 PRINT H$
37 IF H$ = "Z" THEN GO TO 75
38 LET RT = RT + 1
4Ø LET x = CODE H$
45 IF x > 57 THEN LET x = x - 7
5Ø LET y = CODE H$(2)
55 IF y > 57 THEN LET y = y - 7
6Ø POKE RT,x*16 + y - 816
7Ø GO TO 3Ø
75 CLEAR A
```

Appendix B

Glossary of terms

ACCESS TIME The time from the receipt of an address by a memory element to the time for the data from the addressed element to appear at the output.

ACCUMULATOR A register that may be used as the source of one operand and the destination of results from device operations.

ACTIVE HIGH Logic level 1 is the active state.

ACTIVE LOW Logic level 0 is the active state.

ADDRESS The coded location of one byte of memory.

ADDRESSING MODES These are the modes specifying the memory addresses or microprocessor registers to be used with an instruction.

ALPHANUMERIC Refers to alphabetic and numeric characters.

ARCHITECTURE The structure of a system.

ARGUMENT A numeric value used in a program statement.

ARITHMETIC LOGIC UNIT (ALU) An element that can perform several arithmetic and logic functions.

ARITHMETIC SHIFT A shift operation (left or right) in which the value of the sign bit is maintained.

ARRAY A set of variables with each variable identified by a single character representing the array, and subscript numbers representing position within the array, eg B (3,2),B(4,4).

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of seven-bit coded characters, used for information interchange among data processing systems, communications and associated equipment.

ASSEMBLER A computer program that is used to translate mnemonic instructions into their binary equivalent codes and assign memory locations for data and instructions.

ASSEMBLY LANGUAGE A language in which mnemonic

instructions, labels and names are used. These can be translated by an assembler into a machine code program.

ASYNCHRONOUS OPERATION Operation without using a timing reference.

ATTRIBUTES Code of elements of the screen character grid which define colours of paper and ink for displayed characters and the associated display mode.

BACKING STORE A large capacity store such as a cassette tape.

BASIC Beginners All-purpose Symbolic Instruction Code. A high level language, popular for use with microcomputers.

BAUD RATE The serial rate of transmission expressed in bits per second.

BENCHMARK PROGRAM A specimen program that is used to compare and evaluate microprocessors.

BIDIRECTIONAL BUS A bus along which signals may be sent in either direction.

BINARY A number system with base or radix 2.

BINARY CODED DECIMAL (BCD) A code in which each decimal digit is coded using four weighted binary digits.

BIT A **bi**nary digi**t**.

BREAKPOINT A user-specified temporary program halt used in program debugging.

BUG An error in a program.

BUS Parallel conductors connecting two or more devices.

BUS CONTENTION The situation when two or more devices are simultaneously attempting to place data on a data bus.

BYTE The group of eight bits considered by the microprocessor as a binary word.

CARRY BIT The bit used to indicate the occurrence of a carry from the most significant bit in a word.

CHARACTER SET All of the characters assigned to the keyboard.

CHIP The substrate of a single integrated circuit.

CLEAR An input to a device that resets the states to 0.

CLOCK A periodic timing signal used to control a system.

CMOS DEVICES Complementary Metal Oxide Silicon logic elements constructed from *n* or *p* channel field effect transistors to provide lower power consumption and high noise immunity devices.

CONDITIONAL JUMP An instruction that causes a jump to a different part of the program when a given condition is true.

COUNTER A device that changes state after the application of each clock pulse. Normally its output indicates the total number of clock pulses received (up to its capacity).

CRASH Loss of control of a program.

CURRENT LOOP An interface connection that normally uses

20 mA current in the loop to indicate the logic 1 and zero current to indicate logic 0.

CYCLE TIME The time interval for a set of regular operations to be repeated.

DEBOUNCE The conversion of mechanical contact bounce into a clean transition between the two logic states.

DEBUG Removal of programming errors.

DECIMAL ADJUST An operation to convert a binary result into a binary coded decimal result.

DELAY TIME The time between demand signal and appearance of a corresponding output response.

DEMULTIPLEXER A device that directs a time shared input signal to several outputs in order to separate the channels.

DUMP Transfer to a backing store, display file or print hardcopy.

DYNAMIC MEMORY A memory that slowly loses its contents and therefore needs refreshing.

EAROM Electrically Alterable Read Only Memory.

EEROM
E²PROM } Electrically Erasable Read Only Memory.

EPROM Erasable Programmable Read Only Memory.

FIELD PROGRAMMABLE ROM A read only memory that can be programmed by the user.

FIRMWARE Microprograms implemented in read only memory.

FLAG A flip-flop that is normally set to logic 1 after the occurrence of a specified event.

FLOWCHART A graphical representation of a set of micro-computer instructions.

GLITCH An unwanted electrical noise pulse.

HARDWARE The physical devices constituting the micro-computer.

HEXADECIMAL A number system with radix 16. It uses the alphanumeric characters 0 to 9 and A to F.

HIGH LEVEL LANGUAGE A computer language in which the statements represent procedures as opposed to single machine instructions.

INDEX REGISTER A microprocessor register used for memory address modification.

INPUT/OUTPUT (I/O) PORT A connection to the microcomputer that can be programmed for data transfer through an interface.

INSTRUCTION A group of bits used to specify a microprocessor operation.

INSTRUCTION CYCLE The cycle of fetching, decoding and executing an instruction.

INSTRUCTION EXECUTION TIME The time required to fetch, decode and execute an instruction.

INSTRUCTION SET The set of instructions that can be interpreted by the microprocessor.

INTEGER A positive or negative whole number or zero.

INTERFACE The boundary between the computer and its peripherals.

INTERRUPT A microprocessor input that temporarily transfers control from the main program to a separate interrupt routine.

INTERRUPT MASKING A technique that permits the microprocessor to specify if interrupts will be accepted.

INTERRUPT ROUTINE A program that is implemented in response to an interrupt signal.

INVERSE VIDEO Display of a character in which paper and ink colours have been exchanged.

K OF MEMORY 1024 bytes of memory.

LABEL A name given to an instruction or statement in a program in order to identify it.

LATCH A temporary storage element, usually a flip-flop.

LOOP A sequence of instructions that a microcomputer repeats.

MACHINE CODE The language that the microcomputer can interpret directly.

MACHINE CYCLE The basic microprocessor cycle. It is the time required to fetch data from memory or to execute a one-byte instruction.

MASK (a) A bit pattern that separates one or several bits from a group of bits; (b) A photographic plate used in integrated circuit fabrication to define the diffusion patterns.

MASKABLE INTERRUPT An interrupt that can be disabled by the system.

MEMORY An element that can store logic bits.

MEMORY MAP A graphical method of illustrating designated memory sections.

MICROCOMPUTER A microprocessor, memory and interface elements assembled to form a computer.

MICROINSTRUCTION One of the organised sequence of control signals that form instructions at the control level.

MICROPROCESSOR The central processing unit of a microcomputer.

MNEMONIC Symbolic name or abbreviation.

MODEM A modulator/demodulator element that uses a carrier frequency in order to permit communication on a high frequency channel.

MONITOR A simple operating system that permits the user to enter and run programs.

MOS DEVICE Semiconductor elements that use field effect transistors manufactured using Metal Oxide Silicon.

MULTIPLEXER An element that selects one of several inputs and places it on a time shared output.

MULTIPROCESSING Using more than one microprocessor in a single system.

NESTED LOOPS Instruction loops within instruction loops.

NIBBLE A group of four bits.

NMOS N-channel Metal Oxide Silicon.

NON-DESTRUCTIVE READOUT The content of the element can be read without changing the content.

NON-MASKABLE INTERRUPT An interrupt that cannot be disabled by the system.

NON-VOLATILE MEMORY A memory that maintains its content even when the power is removed.

NULL STRING A string containing no characters.

NUMERIC VARIABLE A variable which has a name, and can have a numerical value or numerical expression assigned to it.

OFFSET A number that is added to another number to form an effective address.

ONE'S COMPLEMENT A bit-by-bit complement of a binary number.

OP-CODE The part of a machine code instruction used to specify the operation to be undertaken during the next cycle.

PARITY A one-bit code that is added to a word to make the total number of one-bits in the word even (even parity) or odd (odd parity).

PARITY BIT A status bit that is normally set to logic 1 if the last operation gave a result with (a) even parity, if even parity is used, or (b) odd parity, if odd parity is used.

PEEK An instruction in Basic used to examine the content of a memory byte.

PIXEL A picture element.

PMOS P-channel Metal Oxide Silicon.

POINTER A register or memory location that contains an address.

POKE An instruction in Basic used to write a data byte into a memory location.

POLLING The successive examination of the state of peripherals.

POP Remove an operand from the Stack.

PRIORITY INTERRUPTS Interrupts that can be serviced before others, or may interrupt other interrupt routines.

PROGRAM A sequence of instructions correctly ordered to implement a specific task.

PROGRAM COUNTER A register that holds the address of the next instruction to be executed.

PROGRAMMED INPUT/OUTPUT Input/Output operation implemented under program control.

PROM Programmable Read Only Memory.

PSEUDO-RANDOM Appears to be random, but is not truly random.

PUSH Put an operand on to the Stack.

RAM A memory that can be read and written into. It is referred to as a Random Access Memory.

REAL-TIME CLOCK An element that interrupts a microprocessor at regular time intervals.

REFRESH The process of restoring the content of a dynamic memory.

REGISTER A group of memory cells used to store words within a microprocessor.

ROM Read Only Memory.

RS232 A standard interface for serial communications between computers and peripheral devices.

SCRATCHPAD An area of RAM used for temporary storage of data.

SCROLL The displayed information on the screen is moved upwards and is replaced by new information.

SECOND SOURCE A manufacturer who supplies a device originated from another manufacturer.

SIGN BIT The most significant bit of a data word that indicates the sign of the data. Logic 0 is used to indicate a positive number and logic 1 to indicate a negative number.

SIGNAL CONDITIONING Changing a signal to make it compatible with a specific device.

SOFTWARE Microcomputer programs.

SOFTWARE INTERRUPT An instruction that makes a program jump to a specific address.

STACK A group of RAM memory elements that are normally accessed in a last-in, first-out way.

STACK POINTER A register used to address the next available Stack location.

STATIC MEMORY A memory that does not require refreshing.

STRING A set of characters which forms program text.

STRING VARIABLE A variable, consisting of single alphabetic character followed by $, to which a string can be assigned.

SUBROUTINE A subprogram that can be entered from more than one place in a main program.

SYNCHRONOUS OPERATION Operating at regular intervals of time with respect to a reference time.

SYNTAX The rules of statement structure in a programming language.

SYSTEM VARIABLES These are bytes in RAM identified by a mnemonic and used for specific tasks by the computer.

T-CYCLE Timing cycle of the Z80A microprocessor.

TERMINAL An input/output device used to communicate with a microprocessor system.

TRI-STATE Logic outputs with three possible output levels, namely low, high, and high impedance.

TRUTH TABLE Table showing input and output states for a logic operation.

TTL Transistor–Transistor Logic. This is a very popular bipolar technology used in integrated circuits.

TTL-COMPATIBLE DEVICES These use voltage and current levels within the TTL range and do not need level shifting for interfacing to TTL devices.

TWO'S COMPLEMENT The one's complement of a binary number plus one.

ULA Uncommitted Logic Array – used to implement complicated logic functions.

VOLATILE MEMORY A memory that loses its content when the power is removed.

WORD The group of bits that a microprocessor can manipulate.

WORD LENGTH The number of bits in a microprocessor word.

Appendix C

System variables

The memory area in RAM with addresses 23552 to 23733 is used by the ZX Spectrum Basic interpreter and operating system for storing the *system variables*. A summary of the system variables and their addresses is listed below. Note that for variables of two bytes the first address contains the least significant byte, and the second address the most significant byte.

RAM address	*Variable name*	*Comments*
23552	KSTATE	8 bytes. Used in keyboard reading operation.
23560	LAST K	1 byte. Stores code of most recently depressed key.
23561	REPDEL	1 byte. Time in fiftieths of a second that key may be held down before repeat operation. Preset to 35 but can be POKEd. (Time in sixtieths of a second in USA/Canada.)
23562	REPPER	1 byte. Time delay between successive repeats of key held down. Preset to 5. Delay in fiftieths of a second. (Sixtieths of a second in USA/Canada.)
23563	DEFADD	2 bytes. Stores address of user-defined function arguments if being used. Otherwise it is set equal to zero.

RAM address	*Variable name*	*Comments*
23565	KDATA	1 byte. Stores second byte of colour controls entered from keyboard.
23566	TVDATA	2 bytes. Stores bytes of colour, AT and TAB controls going to TV.
23568	STRMS	38 bytes. Stores addresses of channels attached to streams. Do not POKE this variable, because system may crash.
236Ø6	CHARS	2 bytes. 256 less than address of character set.
236Ø8	RASP	1 byte. Defines time duration of buzz.
236Ø9	PIP	1 byte. Defines time duration of click of keyboard.
2361Ø	ERRNR	1 byte. 1 less than report code.
23611	FLAGS	1 byte. Flags to control Basic. Do not POKE.
23612	TV FLAG	1 byte. Flags concerning TV. Do not POKE.
23613	ERR SP	2 bytes. Address of item on stack used as error return. Do not POKE.
23615	LIST SP	2 bytes. Contains address of return address for automatic listing.
23617	MODE	1 byte. Specifies cursor, either K, L, G, C or E.
23618	NEWPPC	2 bytes. Line to be jumped to.
2362Ø	NSPPC	1 byte. Contains statement number in line to be jumped to.
23621	PCC	2 bytes. Contains line number of statement currently being executed.
23623	SUBPPC	1 byte. Number within line of statement being executed.
23624	BORDCR	1 byte. Border colour attribute.
23625	EPPC	2 bytes. Number of current line.
23627	VARS	2 bytes. Contains address of variables. Do not POKE.
23629	DEST	2 bytes. Contains address of variable in assignment.

RAM address	*Variable name*	*Comments*
23631	CHANS	2 bytes. Contains address of channel data. Do not POKE.
23633	CURCHL	2 bytes. Address of information used currently for input/output. Do not POKE.
23635	PROG	2 bytes. Contains address of Basic program. Do not POKE.
23637	NXTLIN	2 bytes. Contains address of next line in program. Do not POKE.
23639	DATADD	2 bytes. Address of terminator of last data item. Do not POKE.
23641	E LINE	2 bytes. Address of command being keyed in. Do not POKE.
23643	K CUR	2 bytes. Address of cursor.
23645	CH ADD	2 bytes. Address of next character to be interpreted. Do not POKE.
23647	X PTR	2 bytes. Contains address of character following ? cursor.
23649	WORKSP	2 bytes. Contains address of temporary work space. Do not POKE.
23651	STKBOT	2 bytes. Contains address of bottom of microprocessor stack. Do not POKE.
23653	STKEND	2 bytes. Address of start of spare memory area. Do not POKE.
23655	BREG	1 byte. Microprocessor B Register.
23656	MEM	2 bytes. First address of area used for microprocessor memory.
23658	FLAGS2	1 byte. Flags.
23659	DF SZ	1 byte. Number of lines in lower part of screen (including one blank line). Do not POKE.
23660	STOP	2 bytes. Number of top line of program in automatic listing.
23662	OLDPPC	2 bytes. Line number to which CONTinue jumps.
23664	OSPCC	1 byte. Number in line of statement to which CONTinue jumps.
23665	FLAGX	1 byte. Flags.

RAM address	*Variable name*	*Comments*
23666	STRLEN	2 bytes. Length of string type destination in assignment.
23668	T ADDR	2 bytes. Address of next item in syntax table.
2367Ø	SEED	2 bytes. The seed of **RND.** It is set by **RANDOMIZE.**
23672	FRAMES	3 bytes. Frame counter incremented at 20 ms intervals.
23675	UDG	2 bytes. Address of first user-defined graphic symbol.
23677	COORDS	1 byte. x-coordinate of last point plotted.
23678	COORDS	1 byte. y-coordinate of last point plotted.
23679	P POSN	1 byte. Used to track column number of printer position, initialised to 33.
2368Ø	PR CC	1 byte. Least significant byte of address of next position for **LPRINT** to print at.
23681	Not used.	
23682	ECHOE	2 bytes. Used to track column and line number of end of input buffer, initialised to 33 and 24 respectively.
23684	DF CC	2 bytes. Contains address in display file of **PRINT** position.
23686	DF CCL	2 bytes. As DF CC but for lower part of screen.
23688	S POSN	1 byte. Used to track column number for **PRINT** position initialised to 33. Do not POKE.
23689	S POSN	1 byte. Used to track line number for **PRINT** position initialised to 24. Do not POKE.
2369Ø	SPOSNL	2 bytes. As SPOSN but for lower part of screen. Do not POKE.
23692	SCRCT	1 byte. One more than the number of required scrolls before Report code: scroll? is displayed.
23693	ATTR P	1 byte. Used for permanent current colours.

RAM address	*Variable name*	*Comments*
23694	MASK P	1 byte. Used for transparent colours.
23695	ATTR T	1 byte. Used for temporary current colours.
23696	MASK T	1 byte. Used for temporary transparent colours.
23697	P FLAG	1 byte. Flags.
23698	MEMBOT	30 bytes. Scratch pad used for numbers which cannot easily be put on the microprocessor stack.
23728	Not used.	
23730	RAMTOP	2 bytes. Address of last byte of Basic system area.
23732	P-RAMT	2 bytes. Address of last byte of RAM.

Index

ZX Spectrum User's Handbook

The *ZX Spectrum User's Handbook* describes the features and operation of the machine and explains programming in BASIC and machine code to enable you to exploit the Spectrum's features in many practical applications.

The book covers Spectrum hardware and software, and shows you how to interface the computer with peripherals. The sound facilities are described as are the Spectrum's advanced colour capabilities.

Many original programs in BASIC and machine code are included, and the exercises and worked examples will help you gain hands-on experience with the Spectrum.

ISBN 0 408 01323 0

Newnes Microcomputer Books